A High Arithmetic Intensity Krylov Subspace Method Based on Stencil Compiler Programs

Simplice Donfack[1], Patrick Sanan[1], Olaf Schenk[1(✉)], Bram Reps[2], and Wim Vanroose[2]

[1] Institute of Computational Science, Università della Svizzera italiana (USI), Lugano, Switzerland
{simplice.donfack,patrick.sanan,olaf.schenk}@usi.ch
[2] University of Antwerp, Antwerp, Belgium
{bram.reps,wim.vanroose}@uantwerpen.be

Abstract. Stencil calculations and matrix-free Krylov subspace solvers represent important components of many scientific computing applications. In these solvers, stencil applications are often the dominant part of the computation; an efficient parallel implementation of the kernel is therefore crucial to reduce the time to solution. Inspired by polynomial preconditioning, we remove upper bounds on the arithmetic intensity of the Krylov subspace building block by replacing the matrix with a higher-degree matrix polynomial. Using the latest state-of-the-art stencil compiler programs with temporal blocking, reduced memory bandwidth usage and, consequently, better utilization of SIMD vectorization and thus speedup on modern hardware, we are able to obtain performance improvements for higher polynomial degrees than simpler cache-blocking approaches have yielded in the past, demonstrating the new appeal of polynomial techniques on emerging architectures. We present results in a shared-memory environment and an extension to a distributed-memory environment with local shared memory.

Keywords: Stencil compilers · Performance engineering
Krylov methods · Code generation · Autotuning · HPC · CG
Polynomial preconditioning

1 Introduction

Simulation as a discipline relies on increasingly compute-intensive models that require ever more computational resources. Many simulations in science and engineering (e.g., fluid dynamics, meteorology, and geophysics) are based on implicit time-stepping computations using Krylov subspace methods to solve systems based on stencil operators on structured grids. As a result, these applications represent an important high-performance computing software pattern on the current generation of processor architectures [1]. Advancing both stencil

© Springer International Publishing AG, part of Springer Nature 2018
T. Kozubek et al. (Eds.): HPCSE 2017, LNCS 11087, pp. 1–18, 2018.
https://doi.org/10.1007/978-3-319-97136-0_1

grid applications and matrix-free Krylov subspace linear solvers is of utmost importance for implicit time integration applications on structured grids.

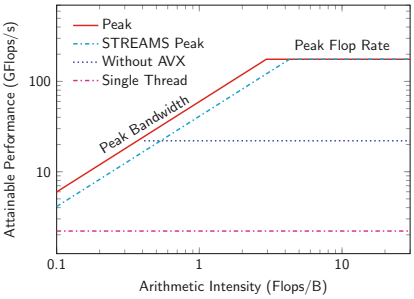

Fig. 1. Roofline model based on the Intel Xeon 2660 v2 Ivy Bridge microarchitecture used in the performance experiments in this paper.

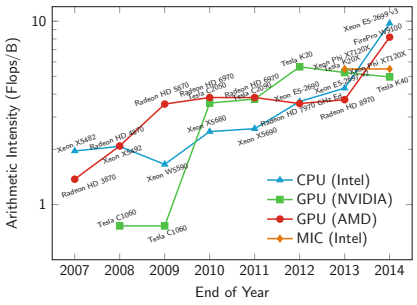

Fig. 2. Double-precision arithmetic intensity to fully utilize the floating-point capabilities of high-performance processing units; reproduced from [28].

Structured grid applications require relatively few arithmetic operations per byte read or written. Vector processor unit performance is thus limited by memory bandwidth rather than compute power, causing structured grid applications to perform poorly on current microarchitectures. The large number of components and machine complexity of future manycore-based architectures will further exacerbate the issue.

Thus, arithmetic intensity, the number of floating point operations executed per byte fetched from main memory, has become a more important metric to evaluate numerical algorithms than the number of required arithmetic operations. Figure 1 shows the relationship between attainable performance and arithmetic intensity on a recent microarchitecture. Below a critical value of arithmetic intensity, performance is limited by memory bandwidth. Stencil-based operations typically fall far below this value. Vectorization (e.g., AVX) allows higher peaks, but requires a higher arithmetic intensity to achieve them. Figure 2 shows the recent increase of the critical value on high-end hardware, plotting the required arithmetic intensity to transition between compute-limited and memory-bandwidth-limited regimes; this trend highlights the need for algorithms to allow simulation codes to adapt.

A given algorithm has a maximum arithmetic intensity, assuming perfect data reuse. Increasing the theoretical maximum is inconsequential if data reuse (with the help of hardware caches) is too suboptimal. Naïve implementations of stencil applications do not come close to achieving this maximum. Blocking strategies are more performant, yet the optimal implementation depends on the details of the cache(s) available. Hence, stencil-code engineering to increase arithmetic intensity has received increased attention in the last few years, evidenced by the appearance of a number of stencil-code programming languages and compilers, such as Pochoir [31], Pluto [6], Patus [11], Modesto [23], and Gscl [5].

In this paper, we make a novel combination of the two approaches to improving performance of stencil-based codes: we simultaneously increase the theoretical maximum arithmetic intensity by reformulating the Krylov solver, and utilize stencil compilers to approach this maximum on current hardware. Both ingredients are essential.

An iteration of a Krylov subspace method includes elementwise vector operations, vector reductions (dot products and norms), and matrix-vector multiplications that are tightly interwoven. This is illustrated in Figs. 3 and 4. In Fig. 3, a 7-point stencil results in the update of a single grid point, after which the stencil is applied repeatedly. In Fig. 4, however, in-between the stencil updates there are additional, interdependent dot products and vector updates. The resulting data dependencies prevent the solver from efficiently applying the subsequent in-place matrix-vector products required both to increase the theoretical arithmetic intensity and to take advantage of advanced stencil engineering techniques ("temporal blocking").

```
1: for t = 1 to nu
2:   for i = 1 to nx
3:     for j = 1 to ny
4:       for k = 1 to nz
5:         U^{t+1}_{i,j,k} ← 6 * U^t_{i,j,k}
6:         −U^t_{i−1,j,k} − U^t_{i+1,j,k}
7:         −U^t_{i,j−1,k} − U^t_{i,j+1,k}
8:         −U^t_{i,j,k−1} − U^t_{i,j,k+1}
```

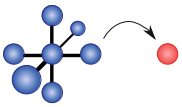

Fig. 3. Stencil example: three-dimensional (3D) Laplacian, 7-point stencil with constant coefficients.

```
1: Initialize X, R and P.
2: for t = 1 to nu do
3:   for i = 1 to nx do
4:     for j = 1 to ny do
5:       for k = 1 to nz do
6:         S^{t+1}_{i,j,k} ← 6 * P^t_{i,j,k}
7:         −P^t_{i−1,j,k} − P^t_{i+1,j,k}
8:         −P^t_{i,j−1,k} − P^t_{i,j+1,k}
9:         −P^t_{i,j,k−1} − P^t_{i,j,k+1}
10:  dot products and vector updates
     with X,S,P,and R
```

Fig. 4. More complicated data dependencies inside the conjugate gradient (CG). Dot products and vector updates that use the results of the dot product limit the arithmetic intensity and make it impossible to use temporal blocking out of the box. See also Algorithm 1.

We use the concept of polynomial preconditioning to reformulate these isolated matrix-vector multiplications with matrix-polynomial-vector multiplications $p_m(A)$ with higher arithmetic intensity. The derived stencil computation can be optimized separately using an efficient stencil computation approach. From a numerical point of view, the search space constructed with a low-order

polynomial $p_m(A)$ might be suboptimal compared to the Krylov subspace based on regular powers of A and therefore needs more cumulative applications of the matrix A. However, due to the fact that data are better reused and that vector units can be used more effectively, overall time to solution can decrease.

This approach differs from s-step methods wherein intermediate vectors are stored and later orthogonalized using TSQR [26]. Our approach communicates only initial and final vectors, increasing the volume of computation and hence the arithmetic intensity. This difference reflects the different objectives of s-step preconditioners (removing the lower bounds on the amount of communication) and our approach (removing upper bounds on the arithmetic intensity achievable).

In this work, we leverage stencil compilers, specific code generation, and autotuning methodology to make Krylov subspace kernels both code- and performance-efficient, in a novel way that critically depends on the combination of polynomial reformulation and the use of advanced stencil compilers. After stencil computation and their challenges are discussed in Sect. 2, in Sects. 3–4, we present a careful performance-oriented reformulation of a polynomial Krylov subspace solver, analyze the performance of automatic stencil code generation in this context, and show scalability on current Intel microarchitecture. Section 5 presents results of the approach in two-dimensional experiments, and we conclude with Sect. 6 which demonstrates the implementation and performance of the technique in a three-dimensional, distributed-memory environment.

2 Node-Level Stencil Performance Engineering

Extracting good performance from stencil-based codes can be challenging because their maximum arithmetic intensity is rather low and thus performance is limited by the bandwidth between memory and compute units. Moreover, if the application requires that the stencil be applied multiple times to each of the spatial grids point, there is potential for performance increase by exploiting temporal data locality, i.e., reuse of the computed data before they are transferred back to memory. As an example, the image in Fig. 3 shows a visualization of the stencil structure of a second-order discretization of the 3D Laplacian.

A fair amount of research has addressed the algorithmic changes and code transformations needed. Loop-tiling approaches [17,22] and compilers using the polyhedral model fall into this category. This is used to determine good tile sizes as well as for autoparallelization. However, only a few stencil compilers can carry out the requested, mostly nontrivial transformations for practical usage. We use the following tools in this work:

– **Patus** [10–12] decouples the algorithm from the scheduling of the computation, i.e., what is vectorized, which loops are tiled or unrolled, and which ones are parallelized. Thus, the "schedule" can be (auto)tuned for the target architecture. The goal is to enable a clean implementation of an algorithm (as opposed to C code cluttered by optimizations) while still delivering the

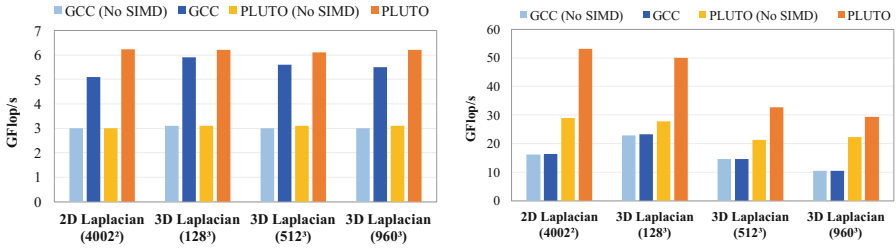

Fig. 5. Impact of AVX vectorization for various stencils using 1 (left) and 10 cores (right) on one socket of an Intel Xeon 2660 v2 Ivy Bridge.

performance of well-optimized implementations. Patus takes advantage of spatial blocking but temporal blocking is not yet available.

– **Pluto** [7] is a research source-to-source (C-to-C) compiler using the polyhedral model. Recently, Pluto was extended to automatically detect iterative stencil computations and to apply both spatial- and time-skewing, allowing concurrent processing of space-time tiles [34]. In this work, we use diamond-tiling loop transformations available in [7] which are more efficient on current microarchitectures. A recent advanced diamond-tiling implementation and performance analysis is also available in [25].

SIMD vectorization has proven to be the key optimization for numerous stencils in all data-centric stencil compiler programs, especially in view of growing SIMD vector width. Using the hardware's SIMD units in an optimal way is critical for performance; even if one trusts the compiler to vectorize scalar code one can observe that explicitly vectorized code using SIMD intrinsics yields significantly better performance. General purpose compilers must be on the safe side and apply vectorization conservatively. Patus and Pluto can generate code using explicit SIMD intrinsics. The benefit is that in the non-unit-stride dimensions no loads are needed which are not aligned at SIMD vector boundaries, as unaligned loads may incur a latency penalty or may not even be supported on other architectures and therefore require a workaround.

However, it is well known that extra care is needed when using SIMD on multiple cores in the presence of limited bandwidth. This is illustrated in Fig. 5, showing AVX vectorization results using Pluto and the GNU GCC compiler (4.9.2). Nonvectorized results are shown as Pluto (No SIMD) and GCC (No SIMD), whereas AVX vectorization has been exploited in the GCC and Pluto cases. Double-precision performance is shown for 5- and 7-point Laplacian stencils on a single socket Intel Xeon 2660 v2 Ivy Bridge node, using 1 and 10 hardware threads. While the automatic vectorization SIMD code generation methods based on the GNU compiler always work well on a single core (with an acceleration of 2.0× for Pluto), it is noticeable that these results cannot automatically be transferred to multiple cores on a socket. As shown in Fig. 6, on multiple cores, the GNU compiler reaches the memory bandwidth limitation of 41.1 GB/s for

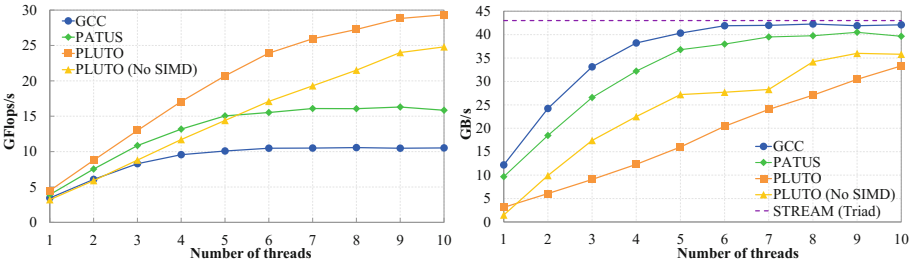

Fig. 6. Performance (left) and bandwidth (right) measurements of a 3D Laplacian of size $n = 960^3$ using a 7-point stencil with constant coefficients.

this memory-bound stencil computation[1]. As a result, there is no improvement from SIMD vectorization. However, if we use techniques such as diamond-tiling loop transformations with Pluto, we reduce the memory-bandwidth pressure and the beneficial impact of SIMD vectorization can be observed even by using up to 10 hardware threads. The trend is also visible in Fig. 6, which shows the performance and measured bandwidth of the Laplacian 7-point stencil with constant coefficients when the number of threads varies. PATUS is highly optimized software and includes optimization techniques beyond autovectorization, so is omitted from Fig. 5. Figure 6 shows that it yields better performance than the naïve approach but does not solve the scalability and bandwidth bottleneck issues. In the rest of the paper, we use PATUS as reference for the best spatial blocking implementation. The main question is how a Krylov subspace method, with much more complicated data dependencies, may benefit from SIMD vectorization, fully utilize all cores without saturating the bandwidth, and take advantage of efficient stencil compilers.

3 Increasing Arithmetic Intensity of Krylov Subspace Methods

Time spent in linear solvers often dominates total execution time in computational science applications, as processor count and problem size increase [9,33]. Krylov subspace methods are commonly used within large-scale, distributed-memory codes. These involve applications of operators, such as the stencils we have considered, interspersed with global reductions (inner products and norms); mitigating the bottleneck created by these operations at extreme scale has prompted the development of methods to hide, as in pipelined Krylov methods [19,20], or avoid, as in s-step methods [13,14], these reductions. Reductions have another detrimental effect beyond the resources required to compute them: they constitute barriers which impose an upper bound on arithmetic intensity, even with perfect data reuse.

[1] We also compared the impact on the vectorization with the Intel *icc* compilers and obtained similar results.

Large, sparse linear systems of the form

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

can be solved with Krylov subspace methods [35] which accept an initial guess $x^{(0)}$ and generate a sequence of iterates $x^{(k)}$ such that the corrections $x^{(k)} - x^{(0)}$ lie in nested subspaces. The iterates satisfy a method-specific optimality criterion and converge toward the solution x. The Krylov subspace of degree k is the space of matrix polynomials of degree less than k applied to $r^{(0)} \doteq b - Ax^{(0)}$, the initial residual:

$$x^{(k)} - x^{(0)} \in \mathcal{K}_k(A, r^{(0)}) = \text{span}\{r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \ldots, A^{k-1}r^{(0)}\}.$$

Depending on the properties of A, the minimization procedure in the search space can be simplified. For example, the CG method [27], shown in Algorithm 1, does not require explicit storage of basis vectors for $\mathcal{K}_k(A, r^{(0)})$ when A is symmetric and positive definite.

Algorithm 1. CG.

```
1:  r ← b − Ax^(0)
2:
3:  p ← r
4:  ρ_old ← ‖r‖^2
5:  for i = 0, 1, ... do
6:      s ← Ap
7:      α ← ρ_old / ⟨s, p⟩
8:      x ← x + αp
9:      r ← r − αs
10:     ρ_new ← ‖r‖^2
11:
12:     β ← ρ_new / ρ_old
13:     p ← r + βp
14:     ρ_old ← ρ_new
15: end for
```

Algorithm 2. Preconditioned CG.

```
1:  r ← b − Ax^(0)
2:  z ← M^−1 r
3:  p ← z
4:  ρ_old ← ⟨r, z⟩
5:  for i = 0, 1, ... do
6:      s ← Ap
7:      α ← ρ_old / ⟨s, p⟩
8:      x ← x + αp
9:      r ← r − αs
10:     ρ_new ← ⟨r, z⟩
11:     z ← M^−1 r
12:     β ← ρ_new / ρ_old
13:     p ← z + βp
14:     ρ_old ← ρ_new
15: end for
```

The residual after k steps, $r^{(k)} \doteq b - Ax^{(k)}$, can be written as a polynomial of A of degree k, applied to the initial residual $r^{(0)}$:

$$r^{(k)} = b - A\left(x^{(0)} + \sum_{j=0}^{k-1} \gamma_j A^j r^{(0)}\right) = r^{(0)} + \sum_{j=0}^{k-1} \gamma_j A^{j+1} r^{(0)} = P_k(A) r^{(0)}.$$

The operations in Algorithm 1 are typically limited by available memory bandwidth and thus cannot take advantage of additional arithmetic operations available in modern devices. Consider a matrix A with n_{nz} nonzero elements. Computing a matrix-vector product, as on line 6 of Algorithm 1, requires reading

n_{nz} matrix elements and at least n elements from the input vector and writing n elements of the output vector. The number of arithmetic operations is $2n_{\text{nz}} - n$.[2] Thus the arithmetic intensity is at most

$$q = \frac{2n_{\text{nz}} - n}{b(n_{\text{nz}} + 2n)} \leq \frac{2}{b}, \tag{2}$$

where b is the number of bytes required to represent a scalar.[3] This represents a small fraction of the arithmetic intensity required to reach peak performance (see Fig. 2).

For matrices described by applying a stencil on a computational grid, matrix entries may be computed instead of read. If no entries need be retrieved from memory, neglecting any floating point operations required to compute the coefficients, the maximum arithmetic intensity of a *matrix-free* operator application can be expressed as

$$q_{\text{matfree}} = \frac{2n_{\text{nz}} - n}{2bn} \leq \frac{n_{\text{nz}}}{bn}. \tag{3}$$

This quantity is characterized by the average number of nonzero entries per row, divided by b. Thus, for some operators, q_{matfree} can be significantly greater than q, but is nonetheless a fixed function of A, and may not be suitable for a given microarchitecture.

However, when m matrix-vector products are done in-place, with the operator A applied repeatedly to v to produce $A^m v$, $m(2n_{\text{nz}} - n)$ operations are performed for a single read of the matrix, one read of the input vector, and one write of the output vector. This results in maximum arithmetic intensities of

$$q = \frac{m(2n_{\text{nz}} - n)}{b(n_{\text{nz}} + 2n)}, \quad q_{\text{matfree}} = \frac{m(2n_{\text{nz}} - n)}{2bn}. \tag{4}$$

These quantities increase linearly with m. The expressions assume that each element of v is read exactly once from main memory, typically impossible, and as such actual attained arithmetic intensity depends on the details of the cache hierarchy and the implementation of the polynomial kernel.

In light of Fig. 1, it is now apparent that one can attempt to improve the performance of Krylov subspace methods by introducing subsequent matrix-vector products instead of isolated matrix-vector products per iteration.

Polynomial preconditioning offers a natural, tunable way to accomplish this, reducing the number of iterations required for convergence of the Krylov subspace method without using additional memory bandwidth. It was first described shortly after CG [29] and became of greater interest with the advent of vector computers [15]. For more, see Sect. 5 of Saad's survey [30]. Multiply (1) on both sides by the polynomial $q_{m-1}(A)$,

$$p_m(A)x = \tilde{b} \doteq q_{m-1}(A)b, \tag{5}$$

[2] n_{nz} multiplies and $n_{\text{nz}} - n$ adds.

[3] For instance, $b = 8$ for double-precision floating point numbers, giving $q < \frac{1}{4}$.

where $p_m(A) \doteq q_{m-1}(A)A$, producing an equivalent linear system with a new operator. Algorithm 2 shows the preconditioned version of the CG method (PCG), where a preconditioning step is explicitly computed on line 11 with the application of a preconditioning operator M^{-1}. This generalized algorithm is useful when the preconditioner is given by a (black box) routine such as algebraic multigrid or ILU decomposition and reduces to Algorithm 1 when $M = I$. However, as $p_m(A)$ is symmetric for symmetric A and can be chosen to be positive definite for positive definite A, polynomial preconditioning only requires standard CG, as in Algorithm 1; line 6 simply computes $s = p_m(A)p_i$ instead of $s = Ap_i$, and b is replaced with $\tilde{b}$. This corresponds to searching for an optimal error correction in a Krylov subspace based on powers of $p_m(A)$:

$$x^{(k)} - x^{(0)} \in \mathcal{K}_k(p_m(A), r^{(0)}) = \text{span}\{r^{(0)}, p_m(A)r^{(0)}, \dots p_m(A)^{k-1}r^{(0)}\}.$$

Note that $\mathcal{K}_k(p_m(A), r_0) \subset \mathcal{K}_{m(k-1)+1}(A, r^{(0)})$ since $p_m(A)r_0$ can be written as a linear combination of the first m vectors of $\mathcal{K}_M(A, r^0)$ for $M > m$. As a result, the residual can be expressed as a polynomial P of the matrix polynomial $p_m(A)$ applied to the initial residual, $r^{(k)} = P(p_m(A))r^{(0)}$.[4]

As discussed in Sect. 4, the polynomial $p_m(A)$ should be defined such that it reduces the number of iterations of the Krylov subspace method. Although the number of arithmetic operations per iteration increases, the overall time to solution is often lower than in the unpreconditioned case.

All other operations in Algorithm 1 are vector updates or dot products with a lower arithmetic intensity than matrix-vector multiplication; reducing the number of total iterations reduces the number of times these must be performed.

The routine to apply the polynomially preconditioned operator $p_m(A)$ can be provided as a black box to a CG solver, allowing it to be optimized with a stencil compiler. This further reduces the demand on memory bandwidth, allowing for arithmetic intensity closer to the ideal in (4) than for a single application of the matrix or a naïvely implemented polynomial kernel. To realize this benefit without the use of a stencil compiler would require expert, hardware-specific implementation. The total number of matrix-vector multiplies with A required to reach a given convergence tolerance typically increases with the use of a polynomially preconditioned system, as opposed to an unpreconditioned one, but time to solution can decrease since the average time per application decreases.

4 Selection and Implementation of the Polynomial Kernel

As test cases, we use simple constant-coefficient Poisson problems in 2 and 3 dimensions,

$$\nabla^2 u(x) = f(x), \quad x \in \mathbb{R}^{\{2,3\}}, \tag{6}$$

discretized with central finite differences, leading to the familiar 5- and 7-point stencils.

[4] This contrasts with s-step Krylov methods [14] that compute iterates in standard Krylov spaces, s basis vectors at a time.

Algorithm 3. Recursive calculation of matrix vector product $w_m = p_m(A)v$.

$\theta = (\lambda_{max} + \lambda_{min})/2$, $\delta = (\lambda_{max} - \lambda_{min})/2$
$\sigma_1 = \theta/\delta$
$\rho_0 = 1/\sigma_1$
$w_0 = 0$, $w_1 = \frac{1}{\theta} Av$
$\Delta w_1 = w_1 - w_0$
for k = 2, . . . , m **do**
    $\rho_{k-1} = 1/(2\sigma_1 - \rho_{k-2})$
    $\Delta w_k = \rho_{k-1} \left[\frac{2}{\delta} A(v - w_{k-1}) + \rho_{k-2} \Delta w_{k-1}\right]$
    $w_k = w_{k-1} + \Delta w_k$
end for

The choice of polynomials p_m is ultimately problem dependent, and has been studied in the literature [2]. A common goal is to minimize, in some norm, $1 - p_m(\lambda)$ over all eigenvalues λ of A. For symmetric systems, this corresponds to the fact that the condition number and clustering of the spectrum provide useful bounds on the convergence [35]. In the case of the simple Laplacian approximations we have chosen, the spectrum is known analytically [8]. It is well-separated over an interval, motivating the choice of Chebyshev polynomials.

If little is known about the spectrum of A, a good polynomial can be constructed based on the Ritz values of A. These can be estimated using a few iterations of CG and calculating the eigenvalues of the tridiagonal Lanczos matrix that contains α_i and β_i.

The recursive definition of Chebyshev polynomials can be used to define a function to apply $p_m(A)$. Algorithm 3 presents such a recursive routine which computes $w_m = p_m(A)v$, where p_m is a scaled and shifted Chebyshev polynomial that maps the spectrum of A around 1. The algorithm comprises only simple operations (and notably no reductions). Thus, when A can be defined as a stencil application, the entire polynomial application is amenable to automatic tiling and cache blocking. This possibility will be exploited in the experiments described in Sects. 5 and 6.

In order to provide data dependencies suitable for tiling with PLUTO, our polynomial kernel implementation used in Sects. 5 and 6 includes the option to use an auxiliary buffer for the vector to which to apply the polynomial.

5 Single-Node Experimental Results

Throughout this section, we compare three solver variants implementing a polynomial stencil application as described in Algorithm 3:

1. a naïve OpenMP implementation obtained by adding `#pragma omp parallel for` directives around the outer spatial loop to parallelize,
2. an improved approach based on the Pluto compiler, and
3. a corresponding implementation using Patus.

Table 1. Relevant hardware information for the Emmy cluster, used for all the experiments in this paper.

Nodes	560	Peak memory bandwidth / socket	59.7 GB/s
Sockets/node	2		
Socket hardware	Xeon 2660v2 (Ivy Bridge)	Peak STREAMS memory bandwidth	41.1 GB/s
Cores/socket	10	DRAM per node	64 GB
Clock frequency	2.2 GHz	L1 cache per core	32 KB instruction + 32 KB data
Double-precision flops/cycle/core	8	L2 cache per core	256 KB
Peak flop rate/socket	176 GFlops/sec	L3 cache per socket	25 MB

5.1 Experimental Environment

All the experiments were performed on the Emmy cluster at RRZE Erlangen, as described in Table 1. We fix the maximum clock rate to 2.2 GHz, disabling "turbo" mode, do not use hyperthreading, and focus on single-socket performance. We use the LIKWID performance tools [32] to bind threads to cores on one socket and run the STREAMS benchmark to measure an attainable memory bandwidth of 41.1 GB/s. The measured bandwidth corresponds to the rate of data transfer between one socket and main memory. The experiments in this section solve a 3D Poisson problem of size $N = 512$ in each dimension, near the maximum size that can be allocated for a single socket. Here, Dirichlet boundary conditions are set in buffer grid points, and loops proceed over interior points only, with a uniform stencil. Code was compiled using *gcc* with `-O3` optimization and AVX enabled. "Degree m" refers to the order of the polynomial $p_m(A)v$.

5.2 Krylov Subspace Method Convergence

Figures 7 and 8 illustrate that the reduction in communication costs results in faster times to solution. Figure 7 shows the convergence history of $\|r^{(k)}\|$ as a function of the number of matrix-vector multiplies for polynomial orders up to $m = 4$. Note that, for each value of m, CG minimizes the error in a different norm, though the residuals can still be used to monitor the convergence.

It is clear that the total number of matrix-vector products before convergence increases with m. This is to be expected, as the iterates lie in subspaces of the full Krylov subspace. Note, however, that the number of dot products to reach convergence (proportional to the number of plot markers) decreases, as does the number of sweeps over the vector. This may allow for lower communication overhead and better cache reuse.

Figure 8 presents the same results as Fig. 7 but plots the residual norm versus time. The polynomial of degree 2 converges faster than the polynomial of degree 1 and, subsequently, convergence time then increases with m. This can be explained by the increasing cost of the matrix-vector products m.

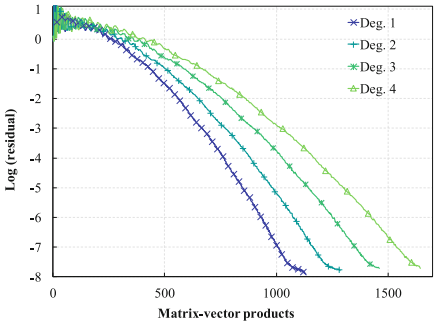

Fig. 7. Polynomial CG convergence as a function of effective matrix-vector multiplies.

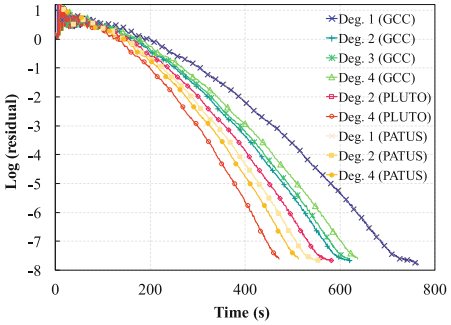

Fig. 8. Polynomial CG convergence using naïve OpenMP, PATUS, and Pluto.

Figure 8 also shows the improvement in terms of convergence rate from using the Pluto compiler. The efficient implementation of the polynomial of degree 2 surpasses all variants of the naïve implementation, and maximal speedup is obtained with a polynomial of degree 4.

Figure 9 shows the time per operation using the naïve OpenMP implementation, Pluto, and PATUS. With the use of the stencil compilers, the matrix-vector product time tends to decrease considerably, resulting in a marked improvement in the time to solution. Using the naïve OpenMP implementation, the total time for the `axpy` and dot product operations decrease as expected with the number of iterations, while the total time for the matrix-vector multiply increases. By implementing the polynomial matrix-vector product $p_m(A)v$ as presented in Algorithm 3 using stencil-based compilers, a speedup of up to 1.5× is attained by using the efficient spatial blocking algorithm Patus, compared to the naïve implementation. The same algorithm using the Pluto temporal blocking algorithm results in speedups of up to 2.6×.

5.3 Scalability

We present the performance scalability of our implementation when the number of threads varies. Figure 10 shows the total time to solution and the measured bandwidth. At around 6 cores, the naïve implementation reaches a measured bandwidth of 40 GB/s, comparable to the maximum attainable bandwidth as measured by the STREAMS benchmark.

By using Patus, we observe a small decrease in the measured bandwidth and a speedup in the time to solution of up to 1.5×, compared to the naïve OpenMP implementation, thanks to the reuse of cache data. Since data reuse is optimized by Patus, each core is likely to use the bus less often. However, each new iteration of the matrix-vector product requires a new load of all the associated data. This problem can be solved using Pluto, thanks to its temporal reuse of data. As shown in the same figure, the Pluto implementation reduces the pressure

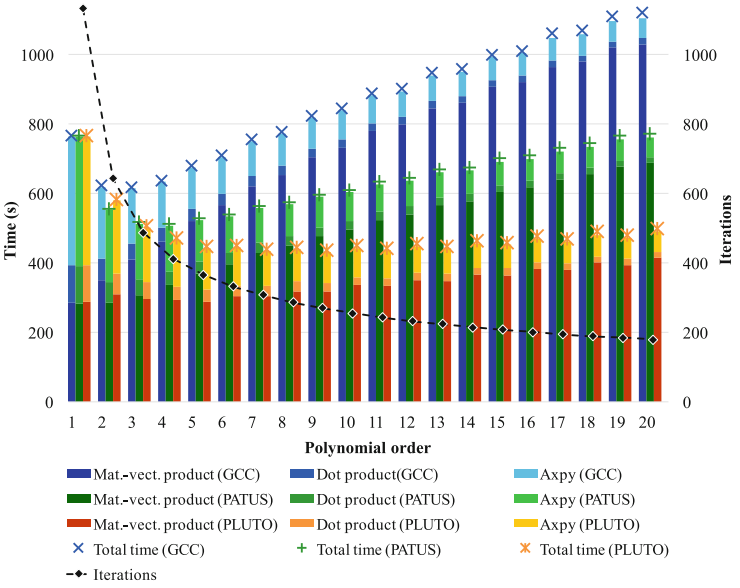

Fig. 9. Time spent for each operation within the OpenMP, PATUS, and PLUTO implementations.

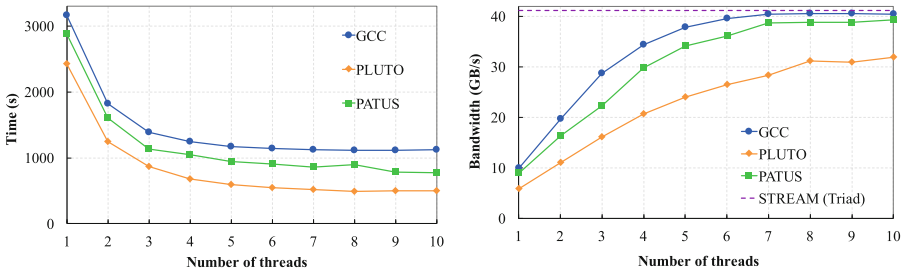

Fig. 10. Performance scalability of our implementation when the number of threads varies. On the left, the time for the solution and, on the right, the measured bandwidth.

on the memory bandwidth when the number of cores increases, and scalability of the algorithm increases considerably. This leads to an implementation up to 2.5× faster than the naïve approach.

6 Multinode Experimental Results

We have shown that the use of a polynomial preconditioner can considerably reduce the iteration count in the CG algorithm. This includes a reduction in the total time spent performing `axpy` and dot product operations.

In this section, we present an implementation and evaluation of our algorithm on multiple nodes as a large-scale, distributed-memory application of our

algorithm in a hybrid MPI/Pthreads environment. In order to reduce the cost of the matrix-vector products, the most time-consuming part of the solver, we use Pluto to increase the arithmetic intensity of the node-level portion of the underlying stencil-based matrix-vector product. We use a distributed version of the CG algorithm where the communications between different processors running on the nodes are performed by MPI, and within each node we use a multithreaded version of the basic operations such as `axpy`, dot product, and matrix-vector product. In order to parallelize the matrix-vector product, we use OpenMP and Pluto. Because of the distributed-memory setting, all the data required to compute the local part of the matrix-vector product at each iteration are stored on the corresponding processor, including redundant "halo" data at the boundaries of per-processor subdomains. Boundaries are computed with specialized, nonoptimized stencils over the appropriate grid points.

For a polynomial of degree m, since we perform m matrix-vector products grouped together, it is necessary to store and exchange a halo region m times the width of the stencil. The extra memory required scales as m times the surface area of the subdomains, $mN^{\frac{d-1}{d}}$, which is small outside of the strong-scaling regime. This approach allows the use of OpenMP and Pluto for the matrix-vector product within each node.

For our tests, we use code leveraging the PETSc library [3,4], linked with the multithreaded BLAS from the Intel MKL. We define our own matrix-free operator which applies the matrix polynomial to data stored on a distributed array (`DMDA`) object, and use it within PETSc's CG. The code used to produce the results in this section, including tests, is available under an open-source license[5]. All results are computed on multiple nodes of the same machine used for the single-node experiments, as described in Table 1. Again, we only use a single socket per node. We compare a version of the code parallelized with OpenMP and a second version which uses Pluto to optimize the polynomial application.

Figure 11 shows the solution time for 2, 4, and 8 nodes when the problem size is fixed at 2048 and the polynomial degree m varies. We observe that while we see strong scaling (and even superlinear strong scaling as the local problem size becomes small enough to fit into the L3 cache), the time to solution using OpenMP tends to increase with the degree of the polynomial. The use of Pluto helps to remove this limitation, allowing further acceleration by increasing the polynomial degree. Figure 12 shows the solution time when the size of the problem increases for a polynomial of degree 10 using 16 and 32 nodes.

We observe that the OpenMP implementation on 32 nodes is up to 2 times faster than on 16 nodes, and that the use of Pluto contributes to the best solution. Figure 13 shows strong scaling with a polynomial preconditioner of degree 10 for a small problem of size 1024. Initial superlinear scaling is observed, likely due to cache effects.

[5] https://bitbucket.org/psanan/polykrylovpetscexample.

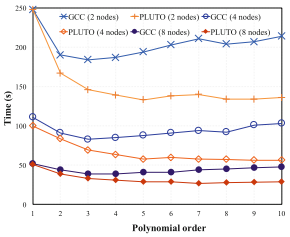

Fig. 11. Solution time for a fixed-size problem, using a polynomial preconditioner on 2–8 distributed-memory nodes.

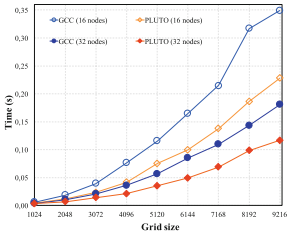

Fig. 12. Time to solution using degree 10 polynomial preconditioning on 16 and 32 distributed-memory nodes.

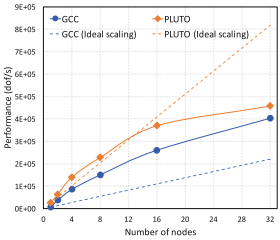

Fig. 13. Strong scaling behavior in time to solution using degree 10 polynomial preconditioning for a small problem of size 1024.

7 Conclusions and Outlook

Krylov subspace solvers use a sparse matrix-vector products, dot products, and vector updates to solve linear system. Each of these building blocks is a low arithmetic intensity operation, limited by available memory bandwidth and implementation's ability to maximize data reuse. As a result, Krylov methods perform poorly on modern processing units that require, high arithmetic intensity algorithms to operate at peak performance, due to widening SIMD registers and increasing shared-memory concurrency.

Modern stencil compilers including temporal blocking can significantly reduce the bandwidth usage of algorithms with repeated application of the same stencil, for example in explicit time evolution.

In this paper we have shown that the same stencil compilers and temporal blocking techniques can also be used to accelerate Krylov methods, although these algorithms have a much more complicated data dependencies. Our insight is that by using polynomial preconditioning in combination with modern stencil compilers, one can simultaneously increase both the maximum and achieved arithmetic intensity, allowing for demonstrable speedup greatly superior to that of direct application on stencil compilers or polynomial preconditioning alone. Indeed, without advanced stencil compilers (or expert, machine-specific tuning), the polynomial approach here would not be beneficial for higher order polynomials, and without the polynomial approach, maximum arithmetic intensity would be limited to that of a single operator application.

The Poisson problem used as a proof of concept here has explicitly known extremal eigenvalues, required to choose a suitable polynomial. For more general operators, these values must be estimated.

A well-known limitation of polynomial preconditioning is that for most operators of interest and fixed m, the number of iterations to convergence of a Krylov method increases with problem size. This may be mitigated in the future as

communication-avoiding preconditioners [21] are developed, and the methods here may also be used with nested or hierarchical solves. A promising area of application is in those situations where a simple, diagonally-preconditioned CG solve, using $O(1)$ iterations, is used within a larger, scalable solver. Examples include multilevel Krylov methods [16]; indeed, an approach of this kind has been shown to be highly effective in an extreme-scale finite element solver, scaling to hundreds of billions of degrees of freedom on hundreds of thousands of cores, using a hierarchy of simply-preconditioned CG solves [18].

We have focused on finite-difference stencils, with regular access patterns allowing efficient optimization. Many problems of interest, such as the application of finite element operators on unstructured meshes, involve more complex access patterns. Stencil compilers for these cases are in their infancy [24], but clear hardware trends will strongly encourage their development, as indeed they will encourage the use and extension of the methods described in this work.

Acknowledgments. We thank Uday Bondhugula for helpful correspondence and upgrades of PLUTO, Karl Rupp for the data in Fig. 2, and Radim Janalik for initial results used in Fig. 5. We acknowledge the Swiss National Supercomputing Center (CSCS) and the University of Erlangen for computing resources. This research has been funded under the EU FP7-ICT project "Exascale Algorithms and Advanced Computational Techniques" (project reference 610741).

References

1. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
2. Ashby, S.F., Manteuffel, T.A., Otto, J.S.: A comparison of adaptive Chebyshev and least squares polynomial preconditioning for Hermitian positive definite linear systems. SIAM J. Sci. Stat. Comput. **13**(1), 1–29 (1992)
3. Balay, S., Abhyankar, S., Adams, M.F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Rupp, K., Smith, B.F., Zampini, S., Zhang, H.: PETSc users manual. Technical report ANL-95/11 - Revision 3.6, Argonne National Laboratory (2015)
4. Balay, S., Gropp, W.D., McInnes, L.C., Smith, B.F.: Efficient management of parallelism in object oriented numerical software libraries. In: Arge, E., Bruaset, A.M., Langtangen, H.P. (eds.) Modern Software Tools in Scientific Computing, pp. 163–202. Birkhäuser Press, Boston (1997). https://doi.org/10.1007/978-1-4612-1986-6_8
5. Bianco, M., Varetto, U.: A generic library for stencil computations. arXiv preprint arXiv:1207.1746 (2012)
6. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: Pluto: a practical and fully automatic polyhedral program optimization system. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008), June 2008. Citeseer, Tucson (2008)
7. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. ACM SIGPLAN Not. **43**(6), 101–113 (2008)

8. Briggs, W.L., Henson, V.E., McCormick, S.F.: A Multigrid Tutorial, 2nd edn. SIAM, University City (2000)
9. Chow, E., Falgout, R.D., Hu, J.J., Tuminaro, R.S., Yang, U.M.: A survey of parallelization techniques for multigrid solvers. In: Parallel Processing for Scientific Computing, vol. 20, pp. 179–201 (2006)
10. Christen, M., Schenk, O., Burkhart, H.: Automatic code generation and tuning for stencil kernels on modern microarchitectures. In: Proceedings of International Supercomputing Conference (ISC 2011), vol. 26, pp. 205–210 (2011)
11. Christen, M., Schenk, O., Burkhart, H.: PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: 2011 IEEE International Conference on Parallel and Distributed Processing Symposium (IPDPS), pp. 676–687. IEEE (2011)
12. Christen, M., Schenk, O., Cui, Y.: PATUS for convenient high-performance stencils: evaluation in earthquake simulations. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 11:1–11:10. IEEE Computer Society Press, Los Alamitos (2012)
13. Chronopoulos, A.T., Swanson, C.D.: Parallel iterative s-step methods for unsymmetric linear systems. Parallel Comput. **22**(5), 623–641 (1996)
14. Chronopoulos, A.T., Gear, C.W.: s-Step iterative methods for symmetric linear systems. J. Comput. Appl. Math. **25**(2), 153–168 (1989)
15. Dubois, P.F., Greenbaum, A., Rodrigue, G.H.: Approximating the inverse of a matrix for use in iterative algorithms on vector processors. Computing **22**(3), 257–268 (1979)
16. Erlangga, Y.A., Nabben, R.: Multilevel projection-based nested Krylov iteration for boundary value problems. SIAM J. Sci. Comput. **30**(3), 1572–1595 (2008)
17. Feautrier, P., Lengauer, C.: The polyhedron model. In: Encyclopedia of Parallel Computing, pp. 1581–1592. Springer, Heidelberg (2011)
18. Fujita, K., Ichimura, T., Koyama, K., Inoue, H., Hori, M., Maddegedara, L.: Fast and scalable low-order implicit unstructured finite-element solver for earth's crust deformation problem. In: Proceedings of the Platform for Advanced Scientific Computing Conference, PASC 2017, pp. 11:1–11:10. ACM, New York (2017)
19. Ghysels, P., Vanroose, W.: Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. Parallel Comput. **40**(7), 224–238 (2014)
20. Ghysels, P., Ashby, T.J., Meerbergen, K., Vanroose, W.: Hiding global communication latency in the GMRES algorithm on massively parallel machines. SIAM J. Sci. Comput. **35**(1), C48–C71 (2013)
21. Grigori, L., Moufawad, S.: Communication avoiding ILU0 preconditioner. SIAM J. Sci. Comput. **37**(2), C217–C246 (2015)
22. Grosser, T., Größlinger, A., Lengauer, C.: Polly - performing polyhedral optimizations on a low-level intermediate representation. Parallel Process. Lett. **22**(4), 1250010 (2012)
23. Gysi, T., Grosser, T., Hoefler, T.: MODESTO: data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS 2015, pp. 177–186. ACM, New York (2015)
24. King, J., Kirby, R.M.: A scalable, efficient scheme for evaluation of stencil computations over unstructured meshes. In: 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12, November 2013

25. Malas, T., Hager, G., Ltaief, H., Stengel, H., Wellein, G., Keyes, D.: Multicore-optimized wavefront diamond blocking for optimizing stencil updates. SIAM J. Sci. Comput. **37**(4), C439–C464 (2015)
26. Mohiyuddin, M., Hoemmen, M., Demmel, J., Yelick, K.: Minimizing communication in sparse matrix solvers. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, p. 36. ACM (2009)
27. Stiefel, E., Hestenes, M.R.: Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards **49**(6) (1952)
28. Rupp, K.: CPU, GPU, and MIC hardware characteristics over time. https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/
29. Rutishauser, H.: Theory of gradient methods. In: Engeli, M., Ginsburg, T., Rutishauser, H., Stiefel, E. (eds.) Refined Iterative Methods for Computation of the Solution and the Eigenvalues of Self-adjoint Boundary Value Problems, pp. 24–49. Springer, Heidelberg (1959). https://doi.org/10.1007/978-3-0348-7224-9_2
30. Saad, Y.: Krylov subspace methods on supercomputers. SIAM J. Sci. Stat. Comput. **10**(6), 1200–1232 (1989)
31. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.-K., Leiserson, C.E.: The Pochoir stencil compiler. In: Proceedings of 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2011), pp. 117–128. ACM (2011)
32. Treibig, J., Hager, G., Wellein, G.: LIKWID: lightweight performance tools. In: Bischof, C., Hegering, H.G., Nagel, W., Wittum, G. (eds.) Competence in High Performance Computing 2010, pp. 165–175. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-24025-6_14
33. U.S. Department of Energy, Office of Advanced Scientific Computing Research. Report on the workshop on Extreme-Scale Solvers: Transition to future Architectures, March 2012. http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/reportExtremeScaleSolvers2012.pdf. Accessed Mar 2013
34. Bondhugula, U., Bandishti, V., Pananilath, I.: Tiling stencil computations to maximize parallelism. In: Proceedings of ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2012), pp. 1–11 (2012)
35. Van der Vorst, H.A.: Iterative Krylov Methods for Large Linear Systems, vol. 13. Cambridge University Press, Cambridge (2003)