# SPECTRAL ELEMENT POISSON PRECONDITIONERS FOR HETEROGENEOUS ARCHITECTURES

BY

MALACHI PHILLIPS

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Doctoral Committee:

Professor Paul Fischer, Chair and Director of Research
Professor Luke Olson
Associate Professor Andreas Kloeckner
Dr. Tzanio Kolev, LLNL
Dr. James Lottes, Google Research

**Abstract**

The solution to the Poisson equation arising from the spectral element discretization of the incompressible Navier-Stokes equation requires robust preconditioning strategies. Two classes of preconditioners prove most effective: geometric $p$-multigrid and low-order refined methods. Low-order refined preconditioners, moreover, require the use of algebraic multigrid to approximate the inverse of the operator. The communication associated with the multigrid coarse-grid solve hinders the parallel scalability of both classes of preconditioners, especially on heterogeneous architectures. To mitigate the coarse-grid solve cost, novel smoothing strategies are considered. The fourth-kind Chebyshev polynomial smoothing proposed by James Lottes is utilized to accelerate additive Schwarz-based smoothers in a geometric $p$-multigrid preconditioner. Through these techniques, we develop geometric $p$-multigrid preconditioners capable of achieving up to an 81% speedup over the state-of-the-art $p$-multigrid preconditioners on the Summit supercomputer. A $p$-multigrid approach with an additive coarse-grid solve specifically designed for heterogeneous architectures is considered. We also propose a hybrid $p$-multigrid and low-order refined preconditioner that improve the time-to-solution by as much as 86% compared to the low-order preconditioner. We demonstrate the effectiveness of these novel approaches on a variety of problems arising from the spectral element discretization of the incompressible Navier-Stokes equations on GPU architectures spanning to $P \geq 1024$ NVIDIA V100 GPUs on Summit.

*To my wife, Miranda*

**Acknowledgments**

Since this dissertation would not have been possible without the support of numerous people, I would like to thank them here.

I would like to thank my advisor, Professor Paul Fischer, for his guidance and support throughout my graduate studies. You have provided many opportunities for me to grow as a researcher. Your conversations and ideas have not only provided me with a deeper understanding of the field, but have also helped me to develop my own research interests and ideas. Your insights from fluid dynamics to solver algorithms have been invaluable. Thank you for your patience and willingness to help me grow as a researcher. Thank you for shaping my interest in fluid dynamics and numerical methods.

A special thanks to my collaborator and coworker, Dr. Stefan Kerkermeier, for his help in the `nekRS` project. You have been a great mentor and friend. Your insights into the implementation of `Nek5000` are vast, and you have been a great resource for me throughout the `nekRS` project. You have provided much inspiration with respect to the development, both algorithmic and implementation-specific, of `nekRS`. I hope that, after this dissertation, we will continue to collaborate on `nekRS` and other projects.

I would like to thank my many collaborators, especially those involved in the CEED project. Professor Tim Warburton, you have proved an oracle of knowledge in writing efficient kernels. Without your help, I would not have the fundamental building blocks for the fast solvers considered in my thesis. Yu-Hsiang Lan, you have been a great collaborator and friend. Without your help, I would not have had the great collection of benchmarks that I have. To all of the other CEED collaborators, I would like to thank you for your help in the development of the CEED project. There are too many to list individually, but I would like to thank you all for your guidance throughout the project. I hope to continue to collaborate with you in the future.

I would like to thank my committee members, Professor Luke Olson, Professor Andreas Kloeckner, Dr. Tzanio Kolev, and Dr. James Lottes. Dr. Tzanio Kolev, I would like to thank you both for your guidance on auxiliary space methods, as well as your contribution in helping lead the CEED project. Dr. James Lottes, I attribute much of my theoretical understanding of multigrid to your work. Your paper and conversations on the fourth-kind Chebyshev polynomial smoother have been invaluable and have influenced the direction of my research. Professor Andreas Kloeckner and Professor Luke Olson, I would like to thank you not only for your insightful comments and suggestions, but for also helping to shape

my interest in numerical methods both during my coursework and through my teaching assistantship with you.

Lastly, I would like to thank my wife, Miranda, for her support and encouragement throughout my graduate studies. I would not have been able to complete this dissertation without her support.

## Table of Contents

# Chapter 1: Introduction

Recent trends in super-computing have seen the rise of heterogeneous computing architectures, wherein the CPU runs in conjunction with an accelerator, such as a GPU. Many of the largest machines in the world, such as Frontier and Summit at Oak Ridge National Laboratory, are heterogeneous architectures; Frontier is currently the largest machine in the world, with a peak performance of $1.194 \times 10^{18}$ floating point operations per second (FLOPs) [1]. Programming for heterogeneous architectures, on the other hand, is a challenge. In the context of physics-based simulations, such as computation fluid dynamics (CFD), communication costs are a primary bottleneck to performance on CPU architectures. On heterogeneous architectures, moreover, the increased throughput offered by the accelerator exacerbates this communication bottleneck, as the relative speed between work in terms of FLOPs and communication increases dramatically. However, the algorithms and discretizations used in CFD simulation codes can be designed to mitigate the effect of communication costs on simulation codes.

Due to their higher arithmetic intensity, high-order methods finite element methods (FEM), including the spectral element method (SEM) [1], are better suited for heterogeneous architectures than low-order methods. In addition, the high-order operators can be cast into a tensor-product sum-factorization form, allowing for the fast application of the operators in general geometries for unstructured problems without requiring the assembly of the operator. In this regard, the methods are *matrix-free* [2]. To effectively utilize the matrix-free methods, however, iterative solution schemes are required to solve the resulting linear systems. Effective preconditioning strategies, therefore, are paramount to the performance of high-order methods.

In this thesis, we consider the use of preconditioning strategies for the SEM Poisson problem on general, unstructured geometries. In the incompressible Navier-Stokes (NS) equations, a pressure Poisson problem is solved at every time instance to enforce the incompressibility constraint. Since our target application is ultimately the solution of the NS equations, the fast solution of the pressure Poisson problem is paramount to overall solution time. In many large scale production simulations for `Nek5000` [3] and `nekRS` [4], the pressure Poisson problem is the dominant cost in the simulation, accounting for as much as 80% of the total solution time. Therefore, to fully realize the potential of SEM incompressible Navier-

[1]The spectral element method (SEM) is a finite element method (FEM) with the combination of a stable Gauss-Lobotto-Legendre (GLL) nodal basis for $\mathbb{Q}_N$ elements with nodal integration. A prominent feature of the SEM is that the resulting mass-matrices are lumped (diagonal). The SEM is a generalization of spectral methods; by taking a single element $E = 1$ and increasing the polynomial degree $p$, the SEM is equivalent to a spectral method.

Stokes simulations on heterogeneous architectures, effective preconditioning strategies are required.

## 1.1 PROBLEM OF INTEREST

In fluid flow simulations, the incompressibility constraint is often used to bypass fast acoustic waves and thereby allow the solution to evolve on a convective time-scale that is most relevant for many engineering problems. This model leads to a Poisson problem for the pressure that is invariably the stiffest sub-step in the time-advancement of the Navier-Stokes (NS) equations. For large 3D problems, which mandate the use of iterative methods, the pressure solve thus typically encompasses the majority of the solution time. Here, several different preconditioning strategies for the Poisson problem in general domains, particularly for discretizations based on high-order SEM, are considered.

The target problem is to solve a sequence of Poisson problems,

$$-\nabla^2 \tilde{u} = \tilde{f} \text{ for } \tilde{u}, \tilde{f} \in \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}. \tag{1.1}$$

The weak formulation is written as: *find* $u^m(\mathbf{x}) \in X_0^N \subset \mathcal{H}_0^1$ *such that*

$$\int_\Omega \nabla v \cdot \nabla u^m \, dV \;\; = \;\; \int_\Omega v \, f^m \, dV \;\; \forall \, v \, \in \, X_0^N, \tag{1.2}$$

where $f^m(\mathbf{x})$ is the data and $u^m(\mathbf{x})$ is the corresponding solution field at some time instant $t^m$, $m = 1, 2, \ldots$ . Here, $\Omega \subset \mathbb{R}^d$ is the computational domain in $d$ (=2 or 3) space dimensions; $\mathcal{H}_0^1(\Omega)$ is the standard Sobolev space comprising functions that vanish on a subset of the boundary, $\partial\Omega_D \subset \partial\Omega$, are square-integrable on $\Omega$, and whose gradient is also square-integrable; and $X_0^N = \text{span}\{\phi_j(\mathbf{x})\}$ is the finite-dimensional trial/test space associated with a Galerkin formulation of the Poisson problem. The discrete problem statement is expressed as $A\underline{u}^m = \underline{b}^m$, where $\underline{u}^m$ is the vector of basis coefficients at $t^m$ and $A$ is the symmetric-positive definite (SPD) matrix with

$$a_{ij} \;\; = \;\; \int_\Omega \nabla \phi_i \cdot \nabla \phi_j \, dV. \tag{1.3}$$

As noted in [5], the SEM is a finite element method (FEM) with the combination of a stable (GLL) nodal basis for $\mathbb{Q}_N$ elements with nodal integration.

For the spectral element (SE) discretization, the Poisson system matrix contains $O(Ep^6)$ non-zeros for $E$ elements with a polynomial degree of $p$ (i.e., approximately $n \approx Ep^3$ un-

knowns.) Through the use of tensor-product-sum factorization, however, the SE matrix-vector product can be effected in only $\approx 7E(p+1)^3$ reads and $12E(p+1)^4$ operations, even in the case of complex geometries [2, 6]. Consequently, the key to fast SE-based flow simulations is to find effective preconditioners tailored to this discretization. Many methods, including geometric $p$-multigrid approaches with point-wise Jacobi and Chebyshev-accelerated Jacobi smoothers [7, 8, 9], geometric $p$-multigrid with overlapping Schwarz smoothers [10, 11, 12], and preconditioning via low-order discretizations [5, 6, 13, 14, 15], have been considered. These are all discussed in further detail in chapter 2.

## 1.2 WHY DO WE NEED FAST POISSON SOLVERS?

As discussed in chapter 1, the Poisson problem is the dominant portion of the solution time in many fluid flow simulations. Therefore, the development of effective preconditioning strategies in `nekRS`, a GPU-accelerated SEM Navier-Stokes solver [4], is crucial. For example, table 1.1 shows that the pressure solve, prior to tuning, encompasses 90% of the overall solution time of the $n = 51B, P = 27648$ V100 GPU `nekRS` simulation of the 352,000 pebble bed geometry on Summit [16]. Even worse, we observe that half of the solution time is spent in the coarse-grid solve, which runs entirely on the CPU! However, by tuning the solver parameters, additional work can be moved onto the smoother through increasing the Chebyshev order, among other things. After tuning, the coarse-grid solve cost is reduced by nearly a factor of 10, just be reducing the number of iterations, and therefore the number of coarse-grid solves required. This translates to over a factor of 2 speedup in the overall solution time.

As we have observed from table 1.1, the design of highly scalable and efficient Poisson preconditioning strategies is imperative for the success of incompressible Navier-Stokes solvers. In this dissertation, we explore several novel preconditioning strategies for the Poisson problem on heterogeneous computing architectures. One major theme of this dissertation is the design of preconditioner algorithms to exploit heterogeneous computing architectures. For example, as we discuss later in this thesis, the coarse-grid solve associated with the lowest level of the geometric $p$-multigrid preconditioner contains an inevitable $\log P$ communication cost scaling, where $P$ is the number of processors. However, this effect is mitigated through designing *robust* preconditioners that require few iterations to converge, thereby limiting the number of times the coarse-grid solve is visited during the linear solve.

While the results in this thesis are specific to linear solvers for the high-order SEM discretization of the pressure Poisson problem, the methods and techniques presented here are applicable to a wider range of problems. Notably, many of the methods discussed in

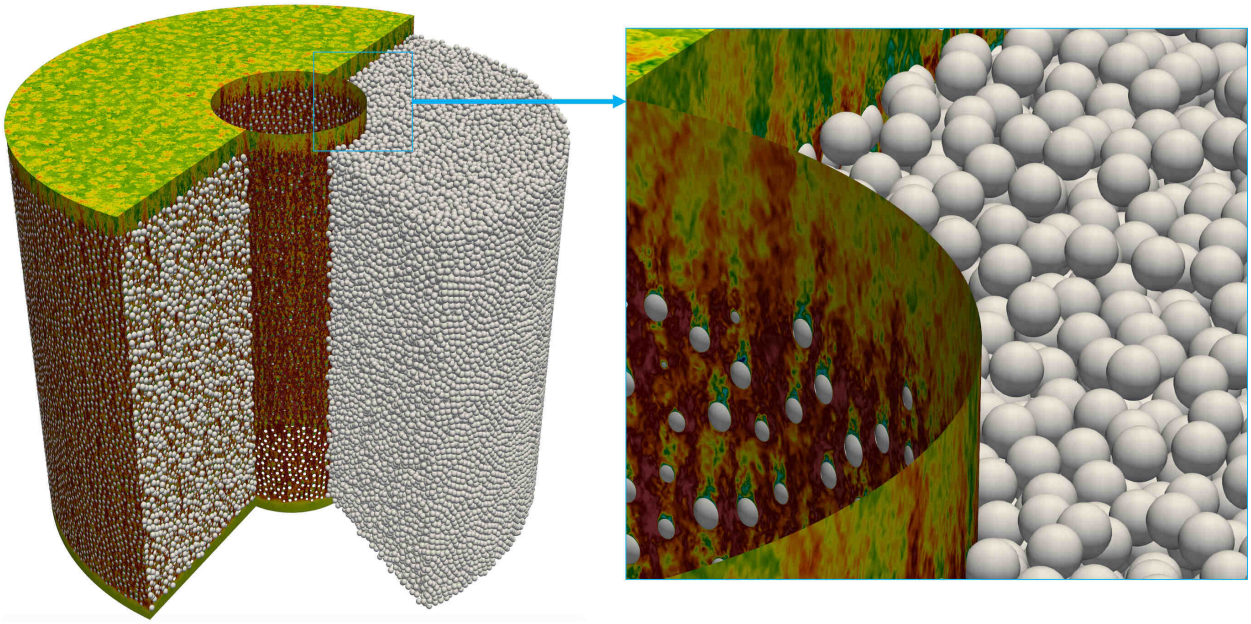


Figure 1.1: 352K pebble geometry from [16], $n = 51B$, $P = 27648$ V100s on Summit.

this thesis are equally applicable to symmetric positive definite linear systems arising from high-order discretizations. While many of the methods discussed in chapter 2 are presented in the context of geometric $p$-multigrid preconditioners, many of the ideas are nevertheless applicable to algebraic multigrid approaches. Further, as we discuss later in chapter 4, the ideas from this thesis are already integrated or nearly integrated into existing geometric and algebraic multigrid preconditioners from popular libraries such as `hypre` [17], `petsc` [18], `Trilinos/MueLu` [19], `LFAToolkit.jl` [20], and `nekRS` [4].

Table 1.1: Run-time statistics for the 352K pebble geometry of fig. 1.1 on $P = 27648$ V100s on Summit [16].

| nekRS Timing Breakdown: n=51B, 2000 Steps | | | | |
|---|---|---|---|---|
| | pre-tuning | | post-tuning | |
| Operation | time (s) | % | time (s) | % |
| computation | 1.19+03 | 100 | 5.47+02 | 100 |
| advection | 5.82+01 | 5 | 4.49+01 | 8 |
| viscous update | 5.38+01 | 5 | 5.98+01 | 11 |
| pressure solve | 1.08+03 | 90 | 4.39+02 | 80 |
| precond. | 9.29+02 | 78 | 3.67+02 | 67 |
| coarse grid | 5.40+02 | 45 | 6.04+01 | 11 |
| projection | 6.78+00 | 1 | 1.21+01 | 2 |
| dotp | 4.92+01 | 4 | 1.92+01 | 4 |

## 1.3 NOVEL CONTRIBUTIONS

In this dissertation, we explore several novel preconditioning strategies for the Poisson problem on heterogeneous computing architectures. Our contributions are summarized as follows:

- We extend the pre-existing literature on Chebyshev polynomial smoothing, as discussed in section 2.7, to accelerate the smoothing properties of a Schwarz-based smoother as detailed in section 2.6.
- Detailed strong/weak scaling studies are conducted to understand the effect of the various preconditioning strategies on the overall solution time, especially *at scale*.
- Theoretical error bound analysis is employed to determine the optimal way of distributing smoothing passes amongst the pre-/post-smoothing phases of the V-cycle, as noted in chapter 4.
- We propose a novel multigrid approach, wherein the coarse-grid solve is treated additively, as discussed in chapter 5. This approach is inspired by the additive multigrid Bramble-Pasciak-Xu (BPX) preconditioner [21] and the additive multigrid ideas of [22]. Notably, this approach is tailored for heterogeneous computing architectures in that the portion of the multigrid scheme that is treated additively is minimal.
- An improvement to the SEMFEM or low-order refined preconditioning strategies through the use of smoothing operations based on the high-order operator is proposed in chapter 6. Theoretical analysis utilizing literature on non-Galerkin coarse grid operators and auxiliary space methods is used to demonstrate the conditioning for this approach can be no worse than the classical SEMFEM preconditioning strategy.

## 1.4 MANUSCRIPT ORGANIZATION

Chapter 1 provides an overview of the problem of interest, the motivation for this work, and the novel contributions of this dissertation. In chapter 2, we aim to inform the reader regarding the spectral element method discretization employed in this work. We further outline the existing state-of-the-art preconditioning methods for preconditioning the Poisson problem in chapter 2, including geometric $p$-multigrid methods, Schwarz-based smoothers, and low-order refined preconditioning. Chapter 3 highlights various programming considerations for heterogeneous architectures, including ensuring performance portability through

run-time GPU kernel selection and algorithms. Further, weak scaling results highlight the implication of the $O(\log P)$ scaling of the coarse-grid solve on the overall solution time. In chapter 4, the Chebyshev polynomial smoothers presented in chapter 2 are expanded to include the novel smoothing techniques based on Chebyshev polynomials of the fourth kind from [23]. These results are further expanded to determine the optimal way of distributing smoothing passes amongst the pre-/post-smoothing phases of the V-cycle. A novel additive coarse-grid solve scheme, based on the seminal BPX preconditioner [21], is presented in chapter 5. This scheme is specifically designed for use in heterogeneous computing environments. Chapter 6 presents an improvement to the SEMFEM preconditioning strategy based on using the low-order operator as an auxiliary coarse-space as described by Xu [24]. The proposed methods in chapters 5 and 6 are tested on a wide variety of problems in chapter 7. Other methods, as described in chapters 2 and 4, are also considered. Finally, the impact from the work in this thesis and future work are summarized in chapter 8.

## Chapter 2: Existing Solver Methods

A brief overview of the existing solver methods considered in this thesis is provided in this chapter. Brief details concerning the spectral element method Poisson discretization are discussed in section 2.1. The first component for the efficient solution of linear systems is an iterative method. Krylov subspace projection (KSP) methods offer *robust* iterative solvers for a wide range of problems. In section 2.2, we discuss the use of flexible preconditioned conjugate gradients and flexible generalized minimal residual methods. Notably, the typical modified Gram-Schmidt orthogonalization employed in the flexible generalized minimal residual method is replaced with classical Gram-Schmidt orthogonalization to reduce the synchronization costs associated with the orthogonalization step. KSPs, moreover, require the use of effective preconditioning strategies. While the Poisson operator is symmetric positive definite (SPD), many of the preconditioners considered in this thesis are *not* symmetric.

As our target application is the time-advancement of the incompressible Navier-Stokes equations, multiple right-hand side solves are required. This presents the opportunity to leverage previous solution states to generate a high-quality initial guess for the current state. We consider the use of Fischer's solution projection method to generate high-quality initial guesses in section 2.3.

With the iterative method and solution projection method in place, we turn our attention to various preconditioning strategies. The high-order spectral element operator has a spectral equivalency to the low-order operator discretized on nodes coinciding with the high-order nodes. This so called SEMFEM equivalency forms the basis of a high-quality preconditioner, as described in section 2.4. Multigrid methods are a popular choice for the solution of linear systems arising from the discretization of partial differential equations. These are described generally in section 2.5. Matrix-free preconditioners build on geometric $p$-multigrid are described in section 2.6, along with the use of Schwarz-based smoothers. Polynomial smoothing based on Chebyshev polynomials of the first-kind are described in section 2.7.

In order to assess the performance of the solver techniques described, we consider moderate size cases in section 2.8. Numerical results for these cases are presented in section 2.9, based on the work in [25].

## 2.1 SPECTRAL ELEMENT DISCRETIZATION

We introduce here basic aspects of the SE Poisson discretization. Consider the Poisson equation in $\mathbb{R}^3$ from eq. (1.1). Boundary conditions for the pressure Poisson equation are either periodic, Neumann, or Dirichlet, with the latter typically applicable only on a small subset of the domain boundary, $\partial\Omega_D$, corresponding to an outflow condition. Consequently, Neumann conditions apply over the majority (or all) of the domain boundary, which makes the pressure problem more challenging than the standard all-Dirichlet case. In this case, the Neumann boundary condition is enforced weakly, and the resultant system contains a non-trivial null-space spanned by the constant vector, $\underline{v} = [1, 1, \ldots, 1]^T$. To correctly handle this null-space, the solution vectors are projected onto the space orthogonal to $\underline{v}$ prior to solving the linear system. Further, any preconditioners must also correctly account for this null-space. For example, the multigrid algorithm later described in section 2.5 must smooth on the lowest level, rather than applying a direct solver.

The SE discretization of eq. (1.1) is based on the weak form: *Find* $u \in X_0^p$ *such that,*

$$(\nabla v, \nabla u)_p = (v, f)_p \;\; \forall v \in X_0^p, \tag{2.1}$$

where $X_0^p$ is a finite-dimensional approximation comprising the basis functions used in the SE discretization, $\phi_j(\mathbf{x})$, $j = 1, \ldots, n$, that vanish on $\partial\Omega_D$ and $(\cdot, \cdot)_p$ represents the discrete $L^2$ inner product based on Gauss-Lobatto-Legendre quadrature in the reference element, $\hat{\Omega} := [-1, 1]^3$. The basis functions allow us to represent the solution, $u$, as $u(\mathbf{x}) = \sum u_j \phi_j(\mathbf{x})$, leading to a linear system of unknown basis coefficients,

$$A\underline{u} = B\underline{f}, \tag{2.2}$$

with respective mass- and stiffness-matrix entries, $B_{ij} := (\phi_i, \phi_j)_p$ and $A_{ij} := (\nabla\phi_i, \nabla\phi_j)_p$ [2].

$\Omega$ is tessellated into non-overlapping hexahedral elements, $\Omega^e$, for $e = 1, \ldots, E$, with isoparametric mappings from $\hat{\Omega}$ to $\Omega^e$ provided by $\mathbf{x}^e(r, s, t) = \sum_{i,j,k} \mathbf{x}^e_{ijk} h_i(r) h_j(s) h_k(t)$, for $i, j, k \in [0, p]$. Each $h_*(\xi)$ is a $p$th-order Lagrange cardinal polynomial on the Gauss-Lobatto-Legendre (GLL) quadrature points, $\xi_j \in [-1, 1]$. Similarly, the test and trial functions $u, v$ are written in local form as $u^e(r, s, t) = \sum_{i,j,k} u^e_{ijk} h_i(r) h_j(s) h_k(t)$. Continuity is ensured across the interface between adjacent elements by enforcing $u^e_{ijk} = u^e_{\hat{i}\hat{j}\hat{k}}$ when $\mathbf{x}^e_{ijk} = \mathbf{x}^e_{\hat{i}\hat{j}\hat{k}}$. From this, a global-to-local degree-of-freedom mapping, $\underline{u} := \{u_l\} \longrightarrow \underline{u}_L := \{u^e_{ijk}\}$, can be

[2] Technically, eq. (2.2) is only true if $B$ is diagonal. The right-hand side is actually $R\bar{B}\bar{\underline{f}}$, where $R$ is a restriction matrix mapping from points to true degrees of freedom, $\bar{B}$ is the unmasked mass matrix, and $\bar{\underline{f}}$ is the unmasked forcing vector. For a more thorough discussion of the SE-discretization, refer to [2, eq. (4.3.24)].

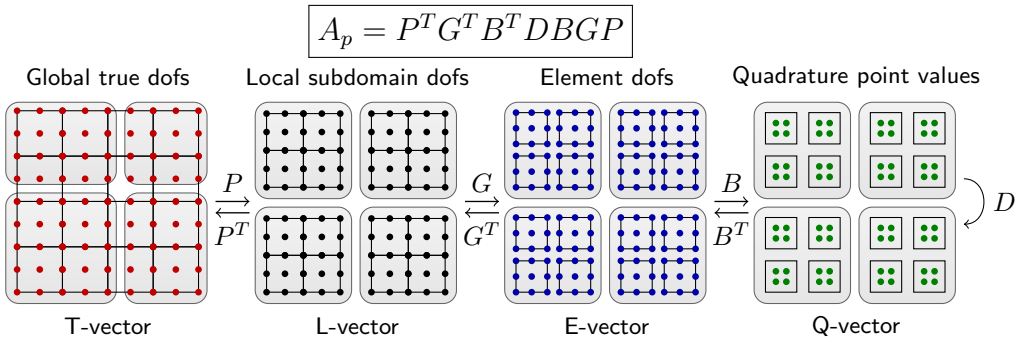



Figure 2.1: Schematic of the high-order operator decomposition used in the `MFEM` and `libCEED` software libraries.

represented by a Boolean matrix $Q$, such that $\underline{u}_L = Q\underline{u}$. The assembled stiffness matrix is then $A = Q^T A_L Q$, where $A_L = \text{block-diag}(A^e)$ comprises the local stiffness matrices, $A^e$. Similarly, $B = Q^T B_L Q$. The SE formulation uses coincident GLL quadrature and nodal points, such that $B^e$ is diagonal. Moreover, $A^e$ is never formed, as it would contain $O(p^6)$ non-zeros in the general case. Rather, the tensor-product-sum factorization [6] allows for $A\underline{u}$ to be evaluated in $O(Ep^4)$ time with $O(Ep^3)$ storage, as described in detail in [2, Section 4.3].

The matrix-free approach described herein can also be expressed in terms of the mathematical description employed by the high-order operator decomposition employed in the `MFEM` [26] and `libCEED` [27] projects. The operator decomposition is illustrated in fig. 2.1. Here, our high-order operator is expressed as $A_p = P^T G^T B^T D B G P$. We note that this operator is never explicitly formed, but rather that function handles compute the *action* of an operator on the vector. In this thesis, we consider the so-called $\mathbb{P}_N - \mathbb{P}_N$ discretization in which the pressure quadrature points and element degrees of freedom are the same. The solution vector $\underline{u}$ can be stored using the unique, true degrees of freedom (i.e., a T-vector in fig. 2.1). Given this storage format, and the fact that quadrature point are element degrees of freedom, we see that the assembled stiffness matrix $A = Q^T A_L Q$ is equivalent to the operator expression $A_p$ with $Q = GP$ and $A_L = B^T D B$.

## 2.2 KRYLOV SUBSPACE SOLVERS

The solution to the large linear systems associated with solving the Poisson equation require the use of Krylov subspace projection methods (KSPs). Since the discrete Poisson operator is symmetric positive definite (SPD), the use of a preconditioned conjugate gradient (PCG) method is the most natural choice. However, the use of a PCG method requires the

use of a symmetric positive definite preconditioner. As we will see in section 2.6 and later section 4.1, however, introducing some asymmetry to the preconditioner can improve the overall convergence rate of the solver. In this section, we will discuss the use of a flexible PCG method and the use of a flexible generalized minimal residual (FGMRES) method to solve the linear systems associated with the Poisson equation. A comparison between the flexible PCG and flexible GMRES methods is presented in section 2.2.4.

### 2.2.1 Flexible Preconditioned Conjugate Gradients

Algorithm 2.1 shows the flexible PCG method from [28]. In this thesis, the termination criteria specified in alg. 2.1 include reaching a specified absolute residual tolerance or a specified relative residual tolerance. The flexible PCG method is the result of a simple modification in computing the $\beta_k$ coefficient. The numerator of the $\beta_k$ coefficient for flexible PCG is computed as $\underline{r}_{k+1}^T(\underline{z}_{k+1} - \underline{z}_k)$ as opposed to $\underline{r}_{k+1}^T\underline{z}_{k+1}$ [29]. Provided that the preconditioner $M^{-1}$ is SPD and constant with respect to the iteration count, $\underline{r}_{k+1}^T\underline{z}_k = 0$, and the update $\beta_k$ coefficient is equivalent. Computing the modified $\beta_k$ term requires storing the additional $\underline{z}_k$ vector and one additional vector operation. However, the total number of dot-products remains the same. In this thesis, two concerns require the use of a *flexible* PCG method: first, the use of non-symmetric preconditioners, and second, the use of a non-constant preconditioner. The use of non-symmetric preconditioners in flexible PCG methods for SPD problems has been justified [28, 30]. Here, the asymmetry in the preconditioner comes from the use of non-symmetric additive Schwarz and restrictive additive Schwarz smoothers in the context of $p$-multigrid preconditioners, see section 2.6. Furthermore, as explained in section 4.1, the use of a non-symmetric distribution of smoothing passes in a multigrid V-cycle preconditioner can enhance the convergence rate of the solver, despite the loss of symmetry. The use of a flexible PCG method also allows for the use of a non-constant preconditioner. This is especially useful when developing a preconditioner that may involve a sub-iteration scheme. For example, the coarse-grid solve required in the multigrid V-cycle preconditioners described in section 2.6 may be performed through an inner PCG solve. For these reasons, we consider the use of a flexible PCG method in this thesis. For brevity, we will refer to the flexible PCG method in alg. 2.1 interchangeably as CG, PCG, or FPCG in the remainder of this thesis.

How quickly the PCG method converges depends on the condition number of the preconditioned operator, $M^{-1}A$. For a fixed, SPD preconditioner, the error at the $k$-th iteration,

$\underline{e}_k = \underline{x}_k - A^{-1}\underline{b}$, satisfies the following error equation [31]:

$$\|\underline{e}_k\|_A \leq 2\left[\frac{\sqrt{\kappa(M^{-1}A)}-1}{\sqrt{\kappa(M^{-1}A)}+1}\right]^k \|\underline{e}_0\|_A. \tag{2.3}$$

Given that a non-symmetric preconditioner is used in conjunction with a flexible PCG method, however, the convergence bound in eq. (2.3) may be violated.

**Algorithm 2.1:** Flexible Preconditioned Conjugate Gradients (FPCG)

**Data:** Initial solution, $\underline{x}_0$, preconditioner $M^{-1}$
**Result:** Solution, $\underline{x}$
$\underline{r}_0 \leftarrow \underline{b} - A\underline{x}_0, \underline{z}_0 \leftarrow M^{-1}\underline{r}_0$, $\underline{p}_0 = \underline{z}_0, k = 0$
**while** *not converged* **do**
  $\alpha_k = \dfrac{\underline{r}_k^T \underline{z}_k}{\underline{p}_k^T A \underline{p}_k}$
  $\underline{x}_{k+1} \leftarrow \underline{x}_k + \alpha_k \underline{p}_k$
  $\underline{r}_{k+1} \leftarrow \underline{r}_k - \alpha_k A\underline{p}_k$
  $\underline{z}_{k+1} \leftarrow M^{-1}\underline{r}_{k+1}$
  $\beta_k = \dfrac{\underline{r}_{k+1}^T(\underline{z}_{k+1} - \underline{z}_k)}{\underline{r}_k^T \underline{z}_k}$
  $\underline{p}_{k+1} \leftarrow \underline{z}_{k+1} + \beta_k \underline{p}_k$
  $k \leftarrow k + 1$
**end**
**return** $\underline{x}_k$

### 2.2.2 Flexible Generalized Minimal Residual Method

Since the use of non-symmetric preconditioners is considered in this thesis, we also consider the use of a flexible generalized minimal residual (FGMRES) method. The FGMRES method is a generalization of the generalized minimal residual (GMRES) method that allows for the use of a non-constant preconditioner. The FGMRES method based on modified Gram-Schmidt (MGS) orthogonalization, as it appears in [32], is shown in alg. 2.2. The number of iteration prior to restarting the FGMRES method is denoted by $m$. Several issues are immediately apparent when comparing FGMRES to flexible PCG. First, the MGS orthogonalization procedure requires $j < m$ dot-products to compute the vector $\underline{v}_{j+1}$. Second, the storage requirement for FGMRES requires the storage of $2m + 1$ vectors. Lastly, this orthogonalization cost can be as large as $O(m^2 n)$ for a $n$-dimensional problem.

**Algorithm 2.2:** Modified Gram-Schmidt Flexible Generalized Minimal Residual Method (MGS-FGMRES)

**Data:** Initial solution, $\underline{x}_0$, preconditioner $M^{-1}$, size of space, $m$
**Result:** Solution, $\underline{x}$
Allocate $(m+1) \times m$ matrix, $H$, with entries $h_{ij} = 0$
$\underline{x} \leftarrow \underline{x}_0$
**while** *not converged* **do**
  $\underline{r}_0 \leftarrow \underline{b} - A\underline{x}, \beta = \|\underline{r}_0\|, \underline{v}_1 = \underline{r}_0/\beta$
  **for** $j = 1, \ldots, m$ **do**
    $\underline{z}_j \leftarrow M^{-1}\underline{v}_j$
    $\underline{w} \leftarrow A\underline{z}_j$
    **for** $i = 1, \ldots, j$ **do**
      $h_{ij} \leftarrow \underline{w}^T\underline{v}_i$
      $\underline{w} \leftarrow \underline{w} - h_{ij}\underline{v}_i$
    **end**
    $h_{j+1,j} \leftarrow \|\underline{w}\|$
    $\underline{v}_{j+1} \leftarrow \underline{w}/h_{j+1,j}$
  **end**
  $Z \leftarrow [\underline{z}_1, \ldots, \underline{z}_m]$
  $\underline{y}_m \leftarrow \arg\min_{\underline{y}} \left\|\beta\underline{e}_1 - H\underline{y}\right\|$
  $\underline{x} \leftarrow \underline{x} + Z\underline{y}_m$
**end**
**return** $\underline{x}$

How can we mitigate the issues with MGS-FGMRES? The first issue regarding the high synchronization cost is resolved through realizing that the orthogonalization step in alg. 2.2 can be replaced with classical Gram-Schmidt (CGS) orthogonalization, as shown in alg. 2.3. This resolves the high synchronization costs associated with MGS orthogonalization, as the dot-product computation can be batched into $j$ simultaneous dot-product operations. The use of CGS orthogonalization, however, introduces potential loss of orthogonality in the computed vectors. Fortunately, this issue, along with the relatively high orthogonalization costs and storage requirement, are mitigated by restricting the number of GMRES iterations prior to a restart to just a few vectors ($m = 15$). We further explore this issue in section 2.2.3. Through investing in robust preconditioners and high-quality initial guesses through solution projection (section 2.3), further, the number of GMRES iterations required to converge to a solution can be reduced. We will refer to the flexible GMRES method with CGS orthogonalization in alg. 2.3 interchangeably as GMRES. In the remainder of this thesis, unless otherwise specified, the flexible GMRES with classical Gram-Schmidt orthogonalization described in the preceding paragraph is utilized as the outer Krylov subspace

solver with $m = 15$ vectors prior to restarting the routine.

**Algorithm 2.3:** Classical Gram-Schmidt Flexible Generalized Minimal Residual Method (CGS-FGMRES)

**Data:** Initial solution, $\underline{x}_0$, preconditioner $M^{-1}$, size of space, $m$
**Result:** Solution, $\underline{x}$
Allocate $(m+1) \times m$ matrix, $H$, with entries $h_{ij} = 0$
$\underline{x} \leftarrow \underline{x}_0$
**while** *not converged* **do**
  $\underline{r}_0 \leftarrow \underline{b} - A\underline{x}, \beta = \|\underline{r}_0\|, \underline{v}_1 = \underline{r}_0/\beta$
  **for** $j = 1, \ldots, m$ **do**
    $\underline{z}_j \leftarrow M^{-1}\underline{v}_j$
    $\underline{w} \leftarrow A\underline{z}_j$
    **for** $i = 1, \ldots, j$ **do**
      $h_{ij} \leftarrow \underline{w}^T\underline{v}_i$  `/* Batch into single j-word all-reduce */`
    **end**
    **for** $i = 1, \ldots, j$ **do**
      $\underline{w} \leftarrow w - h_{ij}\underline{v}_i$
    **end**
    $h_{j+1,j} \leftarrow \|\underline{w}\|$
    $\underline{v}_{j+1} \leftarrow \underline{w}/h_{j+1,j}$
  **end**
  $Z \leftarrow [\underline{z}_1, \ldots, \underline{z}_m]$
  $\underline{y}_m \leftarrow \arg\min_{\underline{y}} \left\|\beta\underline{e}_1 - H\underline{y}\right\|$
  $\underline{x} \leftarrow \underline{x} + Z\underline{y}_m$
**end**
**return** $\underline{x}$

How does the convergence rate of GMRES compare to PCG? Unfortunately, the bound on the convergence of GMRES is more involved than the relatively simple bound for PCG in eq. (2.3). A discussion of the convergence rate of GMRES based on the eigenvalues of the preconditioned operator can be found in [31, Section 6.11.4]. The convergence rate from [31], however, may be overly pessimistic for the case of non-normal operators. Embree notes that bounds based on field of values or pseudo-spectra approaches can provide more useful information regarding GMRES convergence [33].

### 2.2.3 Loss of Orthogonality

In section 2.2.2, we discuss the use of CG-GMRES over MGS-GMRES, preferring the former due to the lower synchronization cost. A well-known limitation of Gram-Schmidt

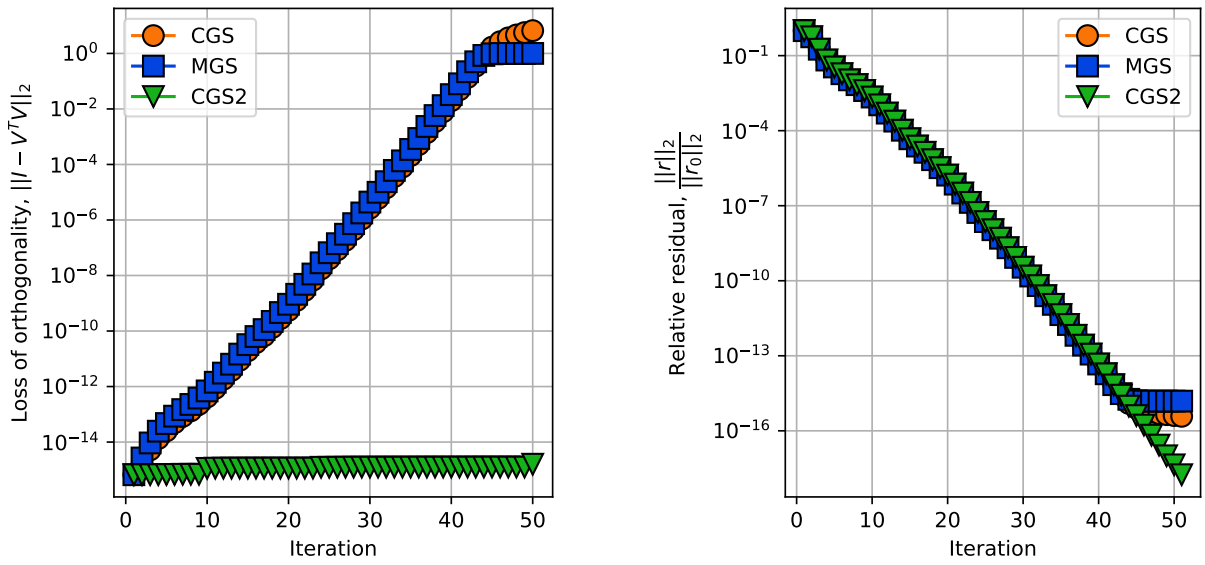



Figure 2.2: CGS-GMRES, MGS-GMRES, and GGS2-GMRES results for $E = 93$ half cylinder problem. Left: loss of orthogonality as a function of iteration count. Right: relative residual as a function of iteration count.

orthogonalization, however, is the potential loss of orthogonality [34, 35] in the computed $\underline{v}_j$ vectors in alg. 2.3. In this section, we demonstrate the loss of orthogonality in the GMRES vectors in the $E = 93$ half cylinder problem later discussed in section 2.4.1. The impact of the loss of orthogonality on the convergence of the GMRES method is also discussed.

To represent a realistic use case, the linear solver is preconditioned with a $1^{st}$-Cheb, ASM(1,1),(7,3,1) multigrid preconditioner (sections 2.6 and 2.7). To facilitate the cancellation errors needed for the loss of orthogonality, a fixed number of $m = 50$ iterations are used without restarting the GMRES method. Defining $V_k$ as the matrix with columns $\underline{v}_j$, $V_k = [\underline{v}_1, \underline{v}_2, \ldots, \underline{v}_k]$, and quantifying the loss of orthogonality as

$$\text{loss of orthogonality} = \left\| I - V_k^T V_k \right\|_2, \tag{2.4}$$

we observe the loss of orthogonality and residual convergence of the various GMRES methods as a function of iteration count in fig. 2.2. In addition to the CGS-GMRES and MGS-GMRES methods, we also consider the use of two-pass classical Gram-Schmidt orthogonalization, referred to as CGS2-GMRES [35]. This method is similar to CGS-GMRES (alg. 2.3), except that the new tentative vector $\underline{v}_{j+1}$ is re-orthogonalized against the previous $j$ vectors in the orthogonalization step. This requires twice the work and synchronization cost as CGS-GMRES, but does not require the $j$ separate synchronizations incurred in the MGS-GMRES

orthogonalization step. The authors note that many alternate orthogonalization routines with relatively low synchronization cost exist, such as iterated Gauss-Seidel GMRES [36].

The left subplot in fig. 2.2 demonstrates that the loss of orthogonality in the GMRES vectors remains relatively low for both CGS-GMRES and MGS-GMRES provided that fewer than $m = 30$ vectors are used prior to restarting the GMRES method. For larger $m$, however, *both* MGS-GMRES and CGS-GMRES experience order unity loss of orthogonalization, with the loss of orthogonality in MGS-GMRES being slightly better than CGS-GMRES. The use of CGS2-GMRES, however, demonstrates that a single additional orthogonalization is sufficient to recover orthogonality despite the large number of GMRES vectors. How does this loss of orthogonality impact the convergence of the GMRES method?

The right subplot in fig. 2.2 demonstrates that the loss of orthogonality in the GMRES vectors has essentially no impact on the convergence of the linear solver. Only when the relative residual is reduced to machine precision does the loss of orthogonality impact the convergence of the GMRES method. In this regime, we observe that CGS2-GMRES is able to further reduce the residual. Despite the loss of orthogonality in the GMRES vectors, the use of either CGS-GMRES or MGS-GMRES is sufficient to converge the linear system to machine precision. Greenbaum and coworkers observe similar results in [37].

### 2.2.4 Loss of Symmetry and GMRES versus CG

One concern with the non-symmetric Schwarz smoothers in section 2.6 and the non-symmetric V-cycle approaches later described in section 4.1 is the loss of symmetry in the preconditioner. This loss of symmetry, however, does not necessarily require the use of GMRES instead of CG as a KSP solver. As noted in section 2.2.1, the flexible PCG algorithm can be used with non-symmetric preconditioners. However, the use of GMRES is preferred.

Further, there are several strategies to mitigate the potential issues with using GMRES, as discussed in section 2.2.2. Using restarted GMRES($m$) bounds the orthogonalization cost for $n$ degrees of freedom to at most $O(m^2 n)$ per iteration. Further, an effective preconditioning strategy can reduce this cost further by ensuring that the number of iterations is small. An additional concern using GMRES($m$) is the $O(m)$ all-reduce operations per iteration. However, by utilizing classical Gram-Schmidt orthogonalization, GMRES($m$) requires only two all-reduce operations per iteration. Concerns over the loss of orthogonality are avoided by keeping $m$ small (e.g., $m = 15$ or $m = 30$). While not used in this current work, Thomas and coworkers demonstrated that iterated Gauss-Seidel GMRES reduces the number of synchronizations to a single all-reduce while preserving backward stability [36].

Kershaw (section 2.8.1) results utilizing $1^{st}$ Cheb, Jacobi(3,3), (7,5,3,1), pMG as a *sym-*

*metric* preconditioner with CG and GMRES(30) as the KSP solvers are shown in table 2.1. For a more complete description of this preconditioner, see sections 2.6 and 2.7. The Kershaw case described in section 2.8.1 uses a $10^{-8}$ relative residual. GMRES(30) produces a lower iteration count than CG. The results presented in table 2.1, however, are meant to illustrate that the time to apply a single GMRES iteration is similar to CG. Consider that, for this case, GMRES(30) is employed, thereby *over-estimating* the cost per iteration for the NS cases in section 2.8.2 wherein GMRES(15) is used. Lastly, the most effective preconditioning strategies utilize Schwarz-based Chebyshev smoothing. The ASM and RAS methods considered herein in section 2.6 are *asymmetric*, and thus the development of a symmetric preconditioning strategy is tabled to pursue a broader set of preconditioners in this thesis.

Table 2.1: Comparison of CG and GMRES(30) for the Kershaw cases using $1^{st}$-Cheb, Jacobi(3,3), (7,5,3,1) as preconditioner.

| $\varepsilon$ | CG iter. | Time per CG iter. | GMRES iter. | Time per GMRES iter. |
|---|---|---|---|---|
| 1 | 20 | $1.26 \times 10^{-2}$ | 9 | $1.27 \times 10^{-2}$ |
| 0.3 | 286 | $1.38 \times 10^{-2}$ | 123 | $1.50 \times 10^{-2}$ |
| 0.05 | 1000 | $1.39 \times 10^{-2}$ | 474 | $1.52 \times 10^{-2}$ |

## 2.3 GENERATING AN INITIAL GUESS THROUGH SOLUTION PROJECTION

During the course of a Navier-Stokes simulation, the linear system associated with the pressure Poisson equation is solved at each time step,

$$A\underline{x}^n = \underline{b}^n, \quad n = 1, \ldots, N. \tag{2.5}$$

Given that the operator, $A$, remains constant or nearly constant over the course of a simulation, high-quality initial guesses for the solution with right-hand side $\underline{b}^n$ can be generated by projecting onto the span of previous solution states.

The exposition here is from Fischer's solution projection scheme in [38]. Let $B_l = \{\tilde{\underline{b}}_1, \ldots, \tilde{\underline{b}}_l\}$ and $X_l = \{\tilde{\underline{x}}_1, \ldots, \tilde{\underline{x}}_l\}$ each be a set of $l < L$ vectors with

$$A\tilde{\underline{x}}_j = \tilde{\underline{b}}_j, \quad j = 1, \ldots, l \tag{2.6}$$

and

$$\tilde{\underline{b}}_j^T \tilde{\underline{b}}_k = \delta_{jk}, \quad j, k = 1, \ldots, l, \tag{2.7}$$

**Algorithm 2.4:** Solution Projection

**Data:** Number of solution vectors, $l$, current right-hand side $\underline{b}^n$, projection state $\{B_l, X_l\}$
**Result:** Solution vector $\underline{x}^n$
**for** $k = 1, \ldots, l$ **do**
  $\alpha_k = \left(\underline{b}^n\right)^T \tilde{\underline{b}}_k$ `/* Batch as single l-word all-reduce */`
**end**
$\tilde{\underline{b}} \leftarrow \underline{b}^n - \sum_{k=1}^{l} \alpha_k \tilde{\underline{b}}_k$
solve $A\tilde{\underline{x}} = \tilde{\underline{b}}$
$\underline{x}^n \leftarrow \tilde{\underline{x}} + \sum_{k=1}^{l} \alpha_k \tilde{\underline{x}}_k$
update $\{B_l, X_l\}$
**return** $\underline{x}^n$

where $\delta_{jk}$ denotes the Kronecker delta with

$$\delta_{jk} = \begin{cases} 0 & \text{if } j \neq k, \\ 1 & \text{if } j = k. \end{cases} \tag{2.8}$$

At time step $n$ with a given right-hand side, $\underline{b}^n$, the solution projection scheme seeks to generate an initial guess in the span of $X_l$ that is as close as possible to the solution of the current linear system. This routine is given in alg. 2.4. While alg. 2.4 utilizes a potentially unstable classical Gram-Schmidt orthogonalization, this allows the $l$ dot-products to be computed in a single $l$-word all-reduce operation, similar to the orthogonalization required in the GMRES algorithm from section 2.2.2. In practice, this is not an issue for the values of $L$ considered in this thesis, with $L = 30$ being the largest value considered. Through using alg. 2.4 to generate an initial guess, the initial residual norm is reduced significantly, typically by a few orders of magnitude.

The solution projection update step required in alg. 2.4 is given in alg. 2.5. For a given user-defined choice of $L$, $2L$ vectors must be stored. However, by ensuring that the vectors are $A$-orthogonal, this storage requirement can be reduced to $L$ vectors at the expense of a single additional matrix-vector product.

The $A$-conjugate solution projection seeks to construct a solution minimizing the $A$-norm error. Given a set of $A$-orthonormal solution vectors $X_l = \{\tilde{\underline{x}}_1, \ldots, \tilde{\underline{x}}_l\}$, we must find $\alpha_k$ coefficients such that

$$\bar{\underline{x}} = \sum_{k=1}^{l} \alpha_k \tilde{\underline{x}}_k \tag{2.9}$$

minimizes the $A$-norm error. Insisting that the vectors in $X_l$ are $A$-orthonormal, the following

**Algorithm 2.5:** Update Projection State

**Data:** Number of solution vectors, $l$, maximum number of vectors, $L$
**Result:** Updated projection state, $\{B, X\}$
**if** $l = L$ **then**
  $\underline{\tilde{b}}_1 \leftarrow A\underline{x}^n / \|A\underline{x}^n\|_2$
  $\underline{\tilde{x}}_1 \leftarrow \underline{x}^n / \|A\underline{x}^n\|_2$
  $l \leftarrow 1$
**else**
  $\underline{\hat{b}} \leftarrow A\underline{\tilde{x}}$
  **for** $k = 1, \ldots, l$ **do**
    $\alpha_k \leftarrow \underline{\hat{b}}^T \underline{\tilde{b}}_k$ `/* Batched as single l-word all-reduce */`
  **end**
  $\underline{\tilde{b}}_{l+1} \leftarrow \left(\underline{\hat{b}} - \sum_{k=1}^{l} \alpha_k \underline{\tilde{b}}_k\right) / \left\|\underline{\hat{b}} - \sum_{k=1}^{l} \alpha_k \underline{\tilde{b}}_k\right\|_2$
  $\underline{\tilde{x}}_{l+1} \leftarrow \left(\underline{\tilde{x}} - \sum_{k=1}^{l} \alpha_k \underline{\tilde{x}}_k\right) / \left\|\underline{\hat{b}} - \sum_{k=1}^{l} \alpha_k \underline{\tilde{b}}_k\right\|_2$
  push $\underline{\tilde{b}}_{l+1}$ onto $\{B\}$
  push $\underline{\tilde{x}}_{l+1}$ onto $\{X\}$
  $l \leftarrow l + 1$

condition holds:

$$\left(\underline{\tilde{x}}_j\right)^T A\underline{\tilde{x}}_k = \delta_{jk}, \quad j, k = 1, \ldots, l. \tag{2.10}$$

This is necessary to simplify the $A$-norm error, as

$$\begin{aligned}\|\underline{x}^n - \underline{\bar{x}}\|_A^2 &= \left(\underline{x}^n\right)^T A\underline{x}^n - 2\sum_{k=1}^{l} \alpha_k \left(\underline{x}^n\right)^T A\underline{\tilde{x}}_k + \underbrace{\sum_{k=1}^{l}\sum_{j=1}^{l} \alpha_k \alpha_j \left(\underline{\tilde{x}}_j\right)^T A\underline{\tilde{x}}_k}_{\left(\underline{\tilde{x}}_j\right)^T A\underline{\tilde{x}}_k = \delta_{jk}} \\ &= \left(\underline{x}^n\right)^T A\underline{x}^n - \sum_{k=1}^{l} \alpha_k \left(\underline{x}^n\right)^T A\underline{\tilde{x}}_k \end{aligned} \tag{2.11}$$

Taking the $A$-norm error in eq. (2.11) and setting it to zero, we have the following result on the $\alpha_k$ coefficients:

$$\begin{aligned}\left(\underline{x}^n\right)^T A\underline{x}^n &= \sum_{k=1}^{l} \alpha_k \left(\underline{x}^n\right)^T A\underline{\tilde{x}}_k \\ &\implies \alpha_k = \left(\underline{\tilde{x}}_k\right)^T A\underline{x}^n, \quad k = 1, \ldots, l \\ &\implies \alpha_k = \left(\underline{\tilde{x}}_k\right)^T \underline{b}^n, \quad k = 1, \ldots, l. \end{aligned} \tag{2.12}$$

The result in eq. (2.12) allows us to construct an orthogonalization procedure similar to alg. 2.4 in alg. 2.6.

**Algorithm 2.6:** $A$-conjugate Solution Projection

**Data:** Number of solution vectors, $l$, current right-hand side $\underline{b}^n$, projection state $\{B_l, X_l\}$
**Result:** Solution vector $\underline{x}^n$
**for** $k = 1, \ldots, l$ **do**
  $\alpha_k = (\underline{\tilde{x}}_k)^T \underline{b}^n$  `/* Batch as single l-word all-reduce */`
**end**
$\underline{\bar{x}} \leftarrow \sum_{k=1}^{l} \alpha_k \underline{\tilde{x}}_k$
solve $A\underline{\tilde{x}} = \underline{\tilde{b}}$
$\underline{x}^n \leftarrow \underline{\tilde{x}} + \underline{\bar{x}}$
update $\{X_l\}$
**return** $\underline{x}^n$

Similar to alg. 2.4, alg. 2.6 requires a mechanism to update the projection space, $X_l$, with a new solution vector. Using the same classical Gram-Schmidt approach as before, this requires only a single additional matrix-vector product and reduces the storage requirement from $2L$ vectors to $L$ vectors. This update procedure is given in alg. 2.7.

**Algorithm 2.7:** Update $A$-conjugate Projection State

**Data:** Number of solution vectors, $l$, maximum number of vectors, $L$
**Result:** Updated projection state, $\{X\}$
**if** $l = L$ **then**
  $\underline{\tilde{x}}_1 \leftarrow \underline{x}^n / \|\underline{x}^n\|_A$
  $l \leftarrow 1$
**else**
  **for** $k = 1, \ldots, l$ **do**
    $\alpha_k \leftarrow (\underline{\tilde{x}}_k)^T A\underline{\tilde{x}}$  `/* Batched as single l-word all-reduce */`
  **end**
  $\underline{\tilde{x}}_{l+1} \leftarrow \left(\underline{\tilde{x}} - \sum_{k=1}^{l} \alpha_k \underline{\tilde{x}}_k\right) / \left\|\underline{\tilde{x}} - \sum_{k=1}^{l} \alpha_k \underline{\tilde{x}}_k\right\|_A$
  push $\underline{\tilde{x}}_{l+1}$ onto $\{X\}$
  $l \leftarrow l + 1$

In `nekRS` and this thesis, the $A$-conjugate solution projection scheme is the default initial guess generator with $L = 10$, unless otherwise specified. At relatively large points per processor, $n/P$, memory requirements often prevent much larger values of $L$ from being considered. However, when assessing the strong-scalability of a solver, the overall memory footprint *per processor* is reduced, allowing for larger values of $L$ to be considered. This,

in turn, can be employed to increase the strong parallel scalability of the solver through a *poly-algorithmic* approach.

While the use of solution projects in this thesis is limited to those proposed by Fischer in [38], others have similarly considered various techniques of leveraging previous solution states to accelerate the solution of linear systems. Parks and coworkers [39] construct a recycling Krylov subspace method, GCRO-DR, to accelerate the convergence of the solver. Further improving the $A$-orthogonality of the solution vectors in alg. 2.6 through Householder reflections or Givens rotations is considered in Christensen's thesis [40]. Notably, two-pass classical Gram-Schmidt achieves similar loss of $A$-orthogonality as single pass modified Gram-Schmidt while still allowing for the $l$ dot-products to be computed in a single $l$-word all-reduce operation. With a special focus on GPU-architectures, Austin and coworkers further expand this idea in [41], including a "rolling" QR-factorization to avoid the need to drop to $l = 1$ solution vectors when the projection space is fully saturated in algs. 2.5 and 2.7. In addition to the novel projection based approaches, the authors also consider stable extrapolation schemes based on least-squares.

## 2.4 PRECONDITIONING BY LOW-ORDER OPERATOR

While the high-order spectral element operator is prohibitively expensive to directly form, as noted in section 2.1, a preconditioner constructed from a low-order discretization with coinciding nodes can be directly formed. In [6], Orszag notes that the inverse of the low-order operator yields bounded conditions numbers which, under certain circumstances, can yield $\kappa(M_F^{-1}A) \sim \pi^2/4$ for second-order Dirichlet problems. To illustrate this point, consider a two-dimensional Poisson problem with Dirichlet boundary conditions discretized with a single spectral element on $\Omega = [0, 1]^2$. The corresponding low-order operator, $A_F$, is constructed using $\mathbb{Q}_1$ finite elements sharing the same nodes as the high-order operator. Details regarding the eigenvalue distribution for varying polynomial orders, $p$, are shown in fig. 2.3.

The order-independent spectral equivalence between the high-order and low-order operators, as demonstrated in fig. 2.3, offers a robust preconditioning strategy for high-order spectral element discretizations. This observation has led to the development of preconditioning techniques based on solving the resulting low-order system [5, 13, 14]. While the inverse of the low-order operator, $A_F^{-1}$, forms a *robust* preconditioner with no dependence on the polynomial order, in practice, the action of $A_F^{-1}$ must be approximated by a preconditioner for $A_F$. Later in this chapter, we will discuss the construction of a preconditioner for $A_F$ based on algebraic multigrid methods. Given a preconditioner, $M_F^{-1}$, for $A_F$, what is the condition number of the resulting preconditioned system, $M_F^{-1}A$? A bound on the

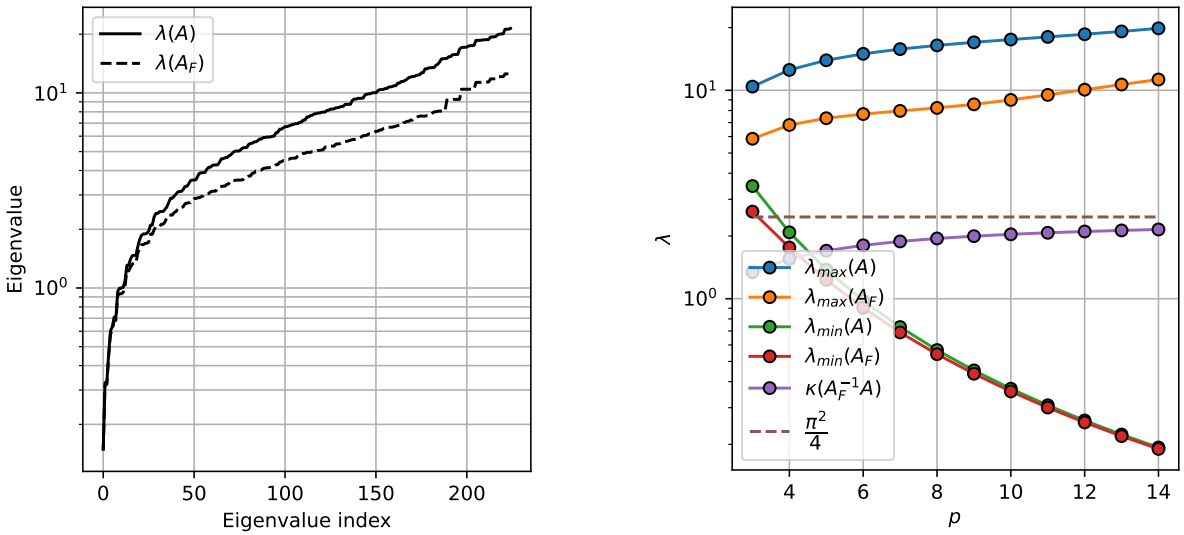



Figure 2.3: Eigenvalue distribution for the Poisson operator, $A$, for a single spectral element in two spatial dimensions with $\Omega := [0,1]^2$. *Left*: Eigenvalue distribution for both the high-order and low-order operators, polynomial degree $p = 16$. *Right*: Min/max eigenvalues, preconditioner condition numbers for varying polynomial orders, $p$. The low-order operator, $A_F$, is constructed using $\mathbb{Q}_1$ finite elements.

condition number of the system is obtained through applying lemma 2.1.

**Lemma 2.1.** Let $A$, $B$, and $C$ be invertible matrices. Then the following condition number bound holds:

$$\kappa(AC) \leq \kappa(AB) \cdot \kappa(B^{-1}C) \tag{2.13}$$

*Proof.* The proof is straightforward, requiring only the definition of the condition number and the Cauchy-Schwarz inequality:

$$\begin{aligned}
\kappa(AC) &= \|AC\| \cdot \left\|(AC)^{-1}\right\| \\
&= \left\|ABB^{-1}C\right\| \cdot \left\|(ABB^{-1}C)^{-1}\right\| \\
&\leq \|AB\| \cdot \left\|B^{-1}C\right\| \cdot \left\|(B^{-1}C)^{-1}\right\| \cdot \left\|(AB)^{-1}\right\| \\
&= \|AB\| \cdot \left\|(AB)^{-1}\right\| \cdot \left\|B^{-1}C\right\| \cdot \left\|(B^{-1}C)^{-1}\right\| \\
&= \kappa(AB) \cdot \kappa(B^{-1}C)
\end{aligned} \tag{2.14}$$

QED.

Substituting $M_F^{-1}$ for $A$, $A_F$ for $B$, and $A$ for $C$ in lemma 2.1, we obtain the following

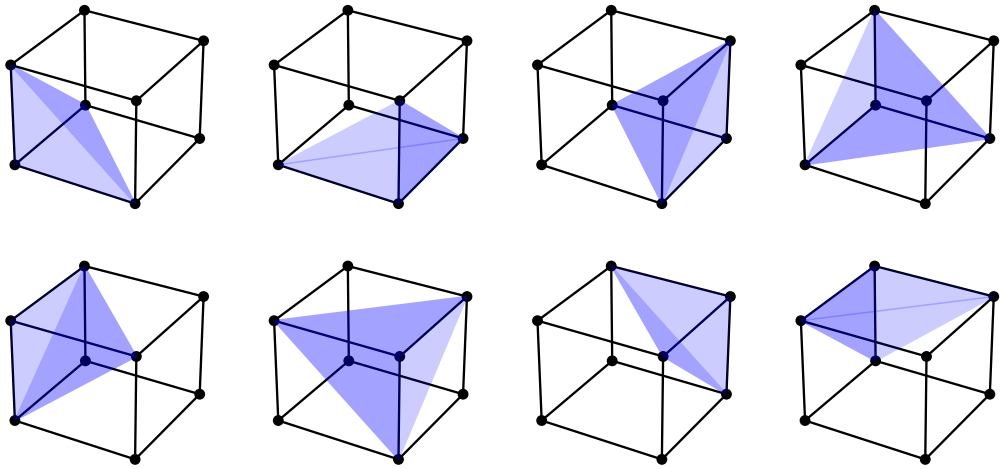


Figure 2.4: Depiction of one tet-per-vertex scheme used to construct low order operator.

condition number bound for the preconditioned system:

$$\kappa(M_F^{-1}A) \leq \kappa(M_F^{-1}A_F) \cdot \kappa(A_F^{-1}A). \tag{2.15}$$

In the preceding paragraph, the importance of constructing a preconditioner, $M_F^{-1}$, for $A_F$ is established. In this work, we consider using the same FEM discretization from Bello-Maldonado and Fischer [14]. The low-order problem is then approximated using a user-defined number of algebraic multigrid (AMG) V-cycles on the GPU through AmgX [42] or boomerAMG [43]. Each of the vertices of a hexahedral element is used to form one low-order, tetrahedral element, as shown in fig. 2.4. This process is repeated for each GLL sub-volume of the high-order hexahedral element. This discretization, based on $\mathbb{P}_1$ finite elements, is found to provide better condition numbers than $\mathbb{Q}_1$ finite elements [14]. This low-order discretization is then used to assemble the sparse operator, $A_F$. The so-called weak preconditioner, $A_F^{-1}$, is used to precondition the system. The following AMG settings are used to approximate the solution to the low order system:

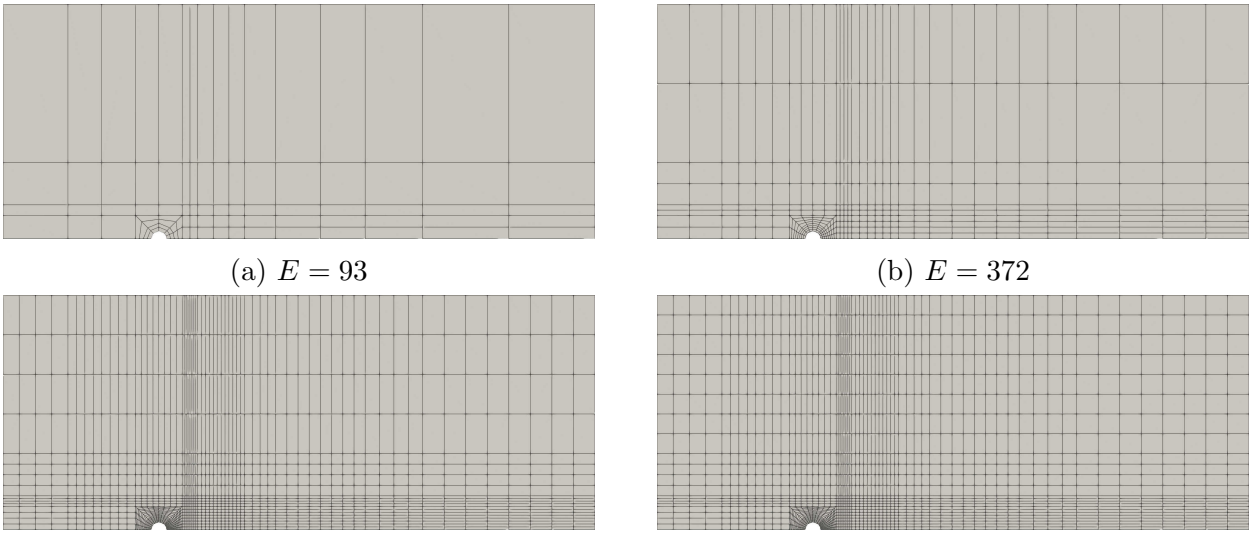
- PMIS coarsening
- 0.25 strength threshold
- Extended + i interpolation ($p_{max} = 4$)
- Damped Jacobi relaxation (0.9)
- One V-cycle for preconditioning
- Smoothing on the coarsest level

This preconditioning strategy is denoted as SEMFEM. *How many AMG V-cycles should be used in the preconditioner?* This concern is addressed in section 2.4.1, where we investigate

the effect of the number of AMG V-cycles on the convergence of the preconditioned system. While increasing the number of V-cycles improves the iteration count, the substantial increase in computational cost is not justified.

### 2.4.1 Effect of Accuracy of Low-Order Operator

The consequence of the condition number bound in eq. (2.15) is that the iteration count for the preconditioned system depends not only on the spectral equivalent between the high- and low-order operators, but also on the condition number of the preconditioned low-order operator. In this section, we investigate the effect of the accuracy of the low-order operator on the convergence of the preconditioned system.

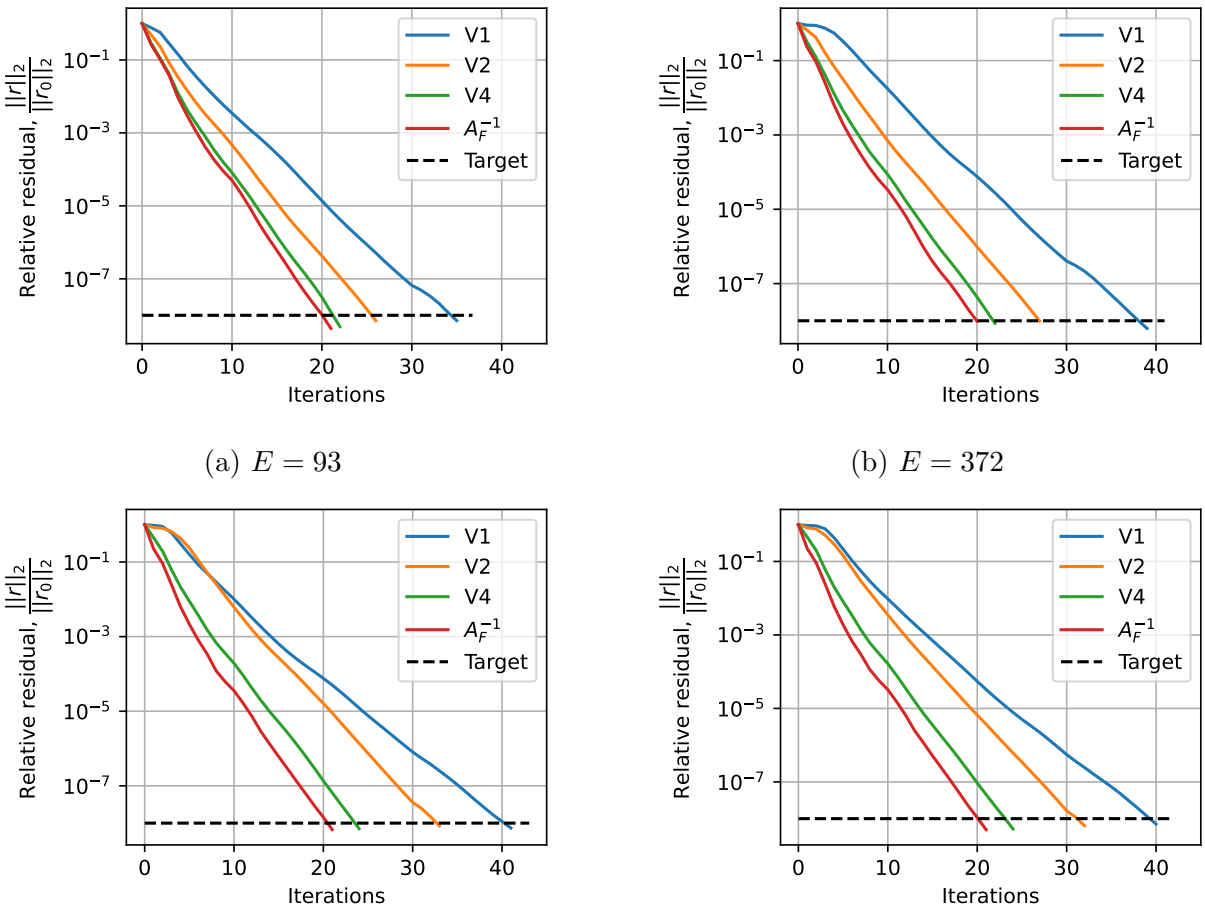


(a) $E = 93$ (b) $E = 372$

(c) $E = 1488$ (d) $E = 1744$

Figure 2.5: Flow past a half-cylinder cases with varying level of $h$-refinement.

We consider the flow past a cylinder problem from [44], as shown in fig. 2.5. Four different levels of $h$-refinement are considered: $E = 93$, $E = 372$, $E = 1488$, and $E = 1744$. A three-dimensional mesh is constructed by extruding the two-dimensional mesh by a single spectral element in the $z$ direction. Dirichlet boundary conditions are applied in the $x$ and $y$ directions, while a periodic boundary condition is applied in the $z$ direction. A random solution vector satisfying the boundary conditions, $\underline{u}$, is used to generate the right-hand side of the linear system, $\underline{b} = A\underline{u}$. GMRES(30) is used to solve the linear system using a termination criteria of a $10^8$ reduction in the residual norm. Four strategies are considered for the low-order operator: 1, 2, and 4 AMG V-cycles to approximate the action of $A_F^{-1}$, and directly applying $A_F^{-1}$.




(a) $E = 93$

(b) $E = 372$

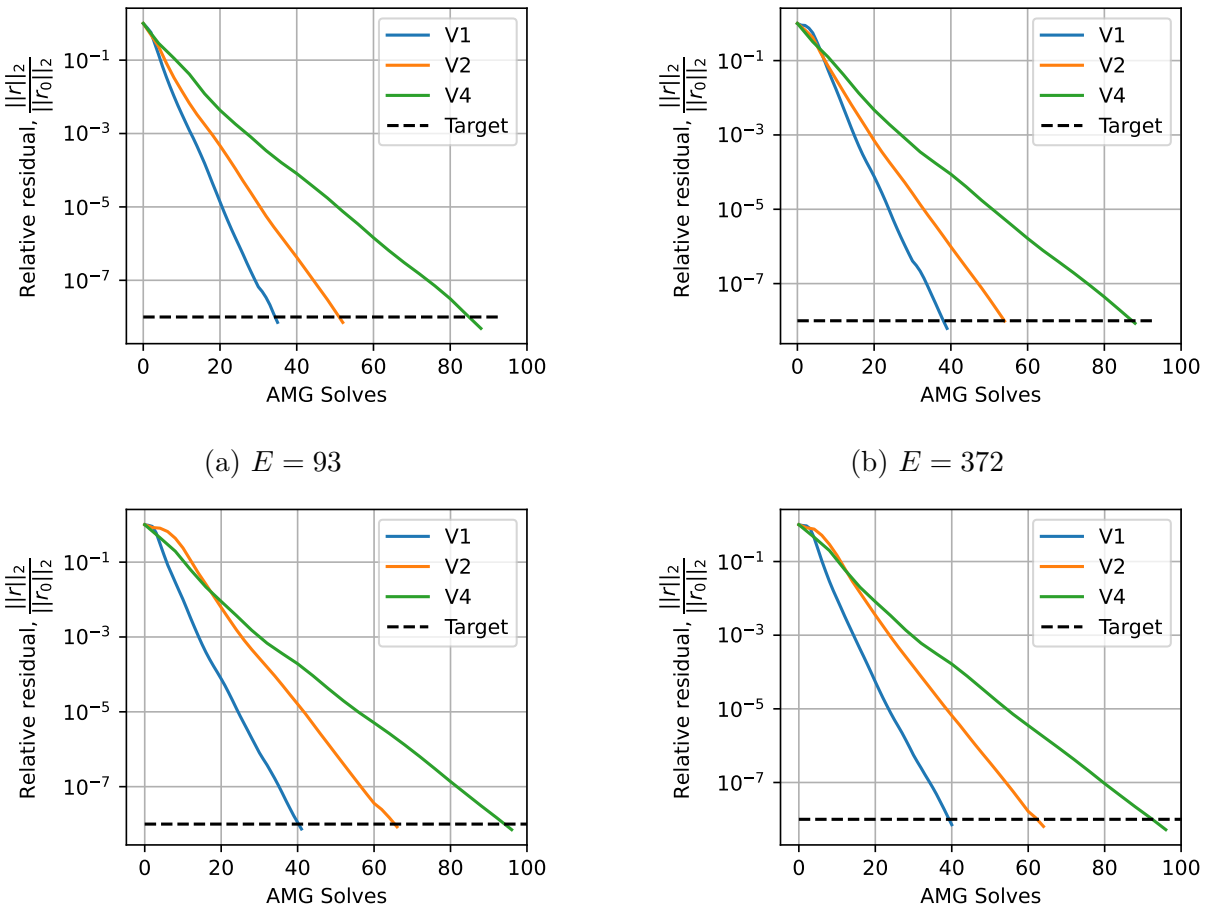
(c) $E = 1488$

(d) $E = 1744$

Figure 2.6: Iteration counts for the SEMFEM preconditioner for the half-cylinder cases from fig. 2.5.

Figure 2.6 shows the residual histories for the four solution strategies described in the prior paragraph. As predicted by the condition number bound in eq. (2.15), the iteration count for the preconditioned system is dependent on the accuracy of the low-order operator. Providing a more accurate approximation of the action of $A_F^{-1}$ results in lower iteration counts. However, this effect has diminishing returns. For example, opting to solve $A_F^{-1}$ directly instead of using 4 AMG V-cycles to approximate the low-order operator results in only a small reduction in the iteration count.

While the results in fig. 2.6 demonstrates that using a more accurate approximation for $A_F^{-1}$ results in lower iteration counts, this comes at the cost of additional AMG V-cycles. Figure 2.7 is similar to fig. 2.6. However, instead of showing the iteration counts, fig. 2.7 shows the number of AMG V-cycles required to reach a given relative residual norm. While




(a) $E = 93$

(b) $E = 372$

(c) $E = 1488$

(d) $E = 1744$

Figure 2.7: AMG V-cycle counts for the SEMFEM preconditioner for the half-cylinder cases from fig. 2.5.

the iteration count is significantly reduced by using 4 AMG V-cycles, for example, the number of AMG V-cycles required to reach a given relative residual norm is significantly higher. As later discussed in section 3.3, the application of a single AMG V-cycle is significantly more expensive than a high-order matrix-vector product. As such, in the remainder of this work, we opt to use a single AMG V-cycle.

## 2.5 MULTIGRID

Let us consider the V-cycle algorithm to solve the SPD matrix $A$. Suppose that levels $j = 0, \ldots, \ell$ are used in the V-cycle, with $A = A_0$. Let us denote the interpolation operator mapping entries from grid $j+1$ to $j$ by $P_{j+1}^{j}$ for $j = 0, \ldots, \ell - 1$. The sequence of matrices

corresponding to each level are typically constructed in a Galerkin fashion, with

$$A_{j+1} = \left(P_{j+1}^{j}\right)^{T} A_j P_{j+1}^{j}, j = 0, \ldots, \ell - 1. \tag{2.16}$$

The multiplicative error propagator for a single V-cycle is given recursively by

$$\begin{aligned} E_j &= I - M_j^{-1} A_j \\ &= G_j' \left( I - P_{j+1}^{j} M_{j+1}^{-1} \left(P_{j+1}^{j}\right)^{T} A_j \right) G_j, j = 0, \ldots, \ell - 1, \end{aligned} \tag{2.17}$$

with $M_\ell^{-1} := A_\ell^{-1}$. $G_j$ and $G_j'$ are the smoother iteration matrices for the pre- and post-smoothing iteration matrices, respectively. For example, $G_j = (I - \omega S_j A_j)^k$ corresponds to $k$ steps of the simple smoothing iteration

$$\left(\underline{x}_{i+1}\right)_j = \left(\underline{x}_i\right)_j + \omega S_j (\underline{b}_j - A_j \left(\underline{x}_i\right)_j), \tag{2.18}$$

where $S_j$ is the smoother for the $j$th level, such as Jacobi with $S_j = \text{diag}(A_j)^{-1}$. $G_j'$ and $G_j$ are not necessarily the same—in fact, we will consider the choice of $G_j' = G_j$ (symmetric post-smoothing) as well as $G_j' = I$ (omitting post-smoothing), among others in Section 4.1.2.

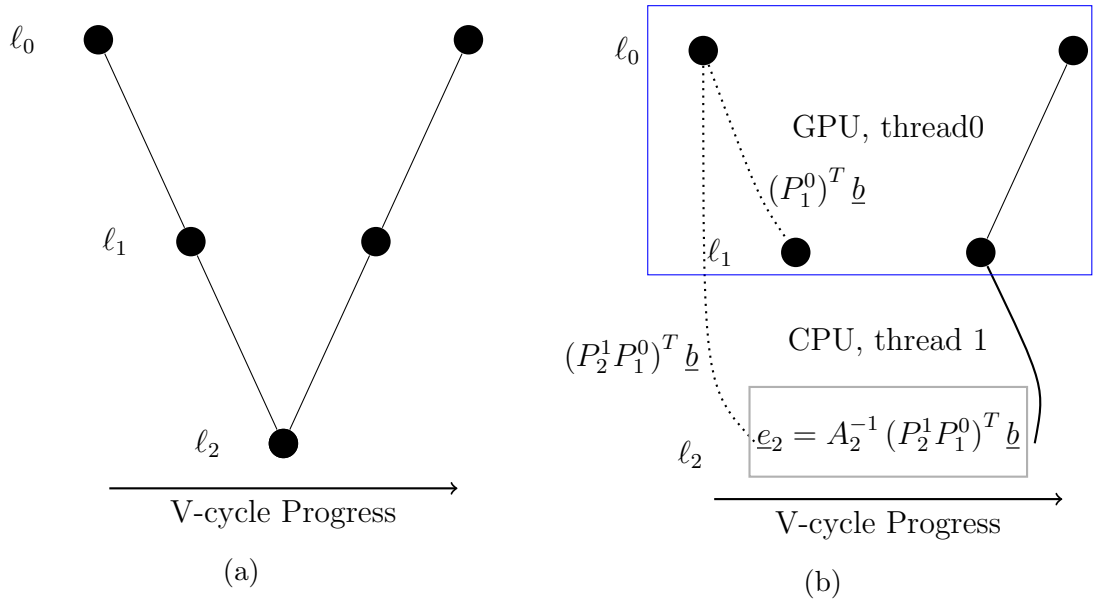



Figure 2.8: Standard, multiplicative multigrid (fig. 2.8a) versus multiplicative multigrid with additive coarse-grid correction (fig. 5.1b). The coarse-grid solve, shown in the dashed box in fig. 5.1b, is performed on the CPU simultaneous with the rest of the multigrid V-cycle on the GPU, shown in the solid box.

The multiplicative V-cycle algorithm discussed in the preceding paragraph is summarized

**Algorithm 2.8:** Multiplicative Multigrid

**Data:** Right-hand side, $\underline{b}$
**Result:** Preconditioned solution, $\underline{z}$
**for** $l = 0, \ldots, \ell - 1$ **do**
  $\underline{x}_l \leftarrow$ `pre-smooth`$(\underline{x}_l, \underline{b}_l)$ `/* e.g., alg. 2.12 */`
  $\underline{r}_l \leftarrow \underline{b}_l - A_l \underline{x}_l$ `/* Residual re-evaluation */`
  $\underline{b}_{l+1} \leftarrow \left(P_{l+1}^l\right)^T \underline{r}_l$ `/* Coarsen */`
**end**
$\underline{x}_\ell = A_\ell^{-1} \underline{b}_\ell$ `/* Coarse-grid Solve */`
**for** $l = \ell - 1, \ldots, 0$ **do**
  $\underline{x}_l \leftarrow \underline{x}_l + P_{l+1}^l \underline{e}_{l+1}$ `/* Prolongate */`
  $\underline{x}_l \leftarrow$ `post-smooth`$(\underline{x}_l, \underline{b}_l)$
**end**
$\underline{z} \leftarrow \underline{x}_0$
**return** $\underline{z}$

in alg. 2.8. Smoother operations `pre-smooth` and `post-smooth` are required. Choices for multigrid smoothers are discussed in greater detail later: the Schwarz smoothers are explained in section 2.6, while polynomial smoothers are expanded upon in section 2.7. Additional results utilizing novel polynomial smoothers developed by Lottes in [23] are explained in section 4.1.

In addition to the multiplicative V-cycle approach described in alg. 2.8, an *additive* V-cycle approach is described in alg. 2.9. Both of these approaches are depicted in fig. 2.8. Notably, alg. 2.9 does not require the additional matrix-vector product associated with the residual re-evaluation after the initial `pre-smooth` pass. The benefits of this approach are two-fold. First, the coarsening operation acts directly on the right-hand side, $\underline{b}_l$, has no dependency on $\underline{x}_l$. This means that several of the operations in alg. 2.9 may be simultaneously overlapped. In this thesis, we consider overlapping the coarse-grid solve, which occurs on the CPU, with the remainder of the cycle, which occurs on the GPU. Second, careful construction of the `pre-smooth` and `post-smooth` operations ensures that *every* matrix-vector product is *projected* as part of the Krylov subspace projection scheme wherein alg. 2.9 is employed as a *preconditioner*. For example, the `pre-smooth` operation can be a single sweep of a Schwarz smoother from section 2.6 or damped Jacobi, which does not require a matrix-vector product since the initial guess $\underline{x}_0$ on a given level is $\underline{0}$. This is an approach utilized in the `Nek5000` hybrid Schwarz multigrid solver [10], and is later compared against other methods in section 2.9. For the `Nek5000`-based solver, the `post-smooth` operation is omitted. We introduce the Schwarz smoothers used in this approach in section 2.6. In chapter 5,

we consider treating only the coarse-grid component as additive, while the remainder of the V-cycle is multiplicative. This is especially beneficial for heterogeneous architectures wherein the coarse-grid solve is already performed on the CPU, allowing for the operations to be performed simultaneously while minimizing additional overhead for concurrent kernel execution [45]. Further, as we discuss in section 3.3, many of the operators employed in the multigrid smoother already overlap communication and computation on the GPU. In this scenario, there is little benefit to overlapping the multigrid levels occurring on the GPU. This is the approach considered in `nekRS` when utilizing the additive multigrid preconditioner. After immediately coarsening the residual, the coarse-grid solve is performed on the CPU while the remainder of the V-cycle is performed on the GPU, as depicted in on the right in fig. 2.8. Since only the coarse-grid solve occurs on the CPU while the remainder of the work is performed on the GPU, *only* the coarse-grid component needs to be additive, while the remainder of the V-cycle is multiplicative. This idea is deferred to chapter 5.

**Algorithm 2.9:** Additive Multigrid

**Data:** Right-hand side, $\underline{b}$
**Result:** Preconditioned solution, $\underline{z}$
**for** $l = 0, \ldots, \ell - 1$ **do**
  $\underline{b}_{l+1} \leftarrow \left(P_{l+1}^{l}\right)^{T} \underline{b}_{l}$ `/* Coarsen */`
**end**
**Thread Block**
  `/* Thread one only uses CPU */`
  **Thread 1**
    $\underline{x}_{\ell} = A_{\ell}^{-1} \underline{b}_{\ell}$ `/* Non-blocking solve on CPU */`
  `/* Thread zero utilizes GPU */`
  **Thread 0**
    **for** $l = 0, \ldots, \ell - 1$ **do**
      $\underline{x}_{l} \leftarrow$ `pre-smooth`$(\underline{x}_{l}, \underline{b}_{l})$ `/* e.g., alg. 2.12 */`
    **end**
`/* Threads synchronized outside of block */`
**for** $l = \ell - 1, \ldots, 0$ **do**
  $\underline{x}_{l} \leftarrow \underline{x}_{l} + P_{l+1}^{l} \underline{x}_{l+1}$ `/* Prolongate */`
  $\underline{x}_{l} \leftarrow$ `post-smooth`$(\underline{x}_{l}, \underline{b}_{l})$
**end**
$\underline{z} \leftarrow \underline{x}_{0}$
**return** $\underline{z}$

For later use in chapter 4, we will need to consider the one-sided V-cycle, which is sufficient

for the analysis of general V-cycles [23, 46]. Similar as before, the "fine-to-coarse"

$$\begin{aligned}(E_{\searrow})_j &= I - (M_{\searrow})_j^{-1} A_j \\ &= \left(I - P_{j+1}^j \left(M_{\searrow}\right)_{j+1}^{-1} \left(P_{j+1}^j\right)^T A_j\right) G_j, j = 0, \ldots, \ell - 1,\end{aligned} \tag{2.19}$$

and "coarse-to-fine"

$$\begin{aligned}(E_{\nearrow})_j &= I - (M_{\nearrow})_j^{-1} A_j \\ &= G_j' \left(I - P_{j+1}^j \left(M_{\nearrow}\right)_{j+1}^{-1} \left(P_{j+1}^j\right)^T A_j\right), j = 0, \ldots, \ell - 1,\end{aligned} \tag{2.20}$$

error propagators are defined, with $(M_{\searrow})_\ell^{-1} = (M_{\nearrow})_\ell^{-1} = A_\ell^{-1}$. Let $E_{\searrow} = (E_{\searrow})_0$ and $E_{\nearrow} = (E_{\nearrow})_0$. The general V-cycle, therefore, is the product of the "fine-to-coarse" and "coarse-to-fine" error propagators,

$$E_V = E_{\nearrow} E_{\searrow}. \tag{2.21}$$

## 2.6 $P$-MULTIGRID AND SCHWARZ-BASED SMOOTHERS

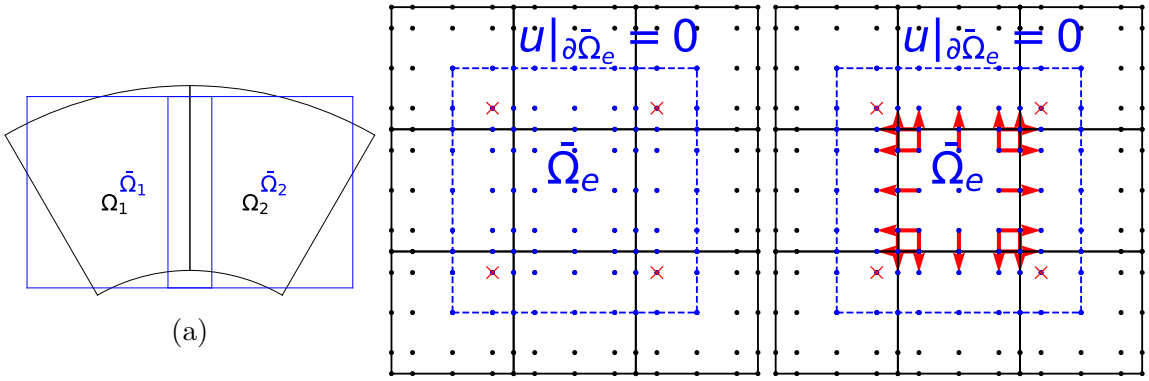
A popular class of high-order FEM Poisson preconditioners is based on geometric $p$-multigrid [7, 9, 11, 12, 47]. Borrowing the notation from section 2.5, each level in $j = 0, \ldots, \ell$ is a geometric level corresponding to a mesh with polynomial degree $p_j$. Typical level schedules, for example, would be the choice of $(p_0, p_1, p_2) = (7, 3, 1)$ for a $p = 7$ problem. The prolongation operators, $P_{j+1}^j$, are applied matrix-free using the tensor-product-sum factorization. To keep the $p$-multigrid approach matrix-free, the sequence of matrices as described in eq. (2.16) cannot be formed. Further, applying the Galerkin operators in eq. (2.16) in a matrix-free manner would require applying the operator at the finest level. This implies that there is no cost reduction in the coarse levels. As such, each operator in eq. (2.16) is chosen as the matrix-free operator corresponding to the given polynomial order of the level, and is therefore non-Galerkin.

As discussed in section 2.5, the V-cycle requires the application of a smoother. These range from point-wise and Chebyshev-accelerated Jacobi [7, 9], which are discussed in section 4.1, to SE-based Schwarz methods, such as that presented in [10, 12]. The SE-based additive Schwarz method (ASM) solves a local Poisson problem on subdomains that are extensions of the spectral elements. The formal definition of the ASM preconditioner (or, in this case,

pMG smoother) is

$$S_{ASM}\underline{r} = \sum_{e=1}^{E} W_e R_e^T \bar{A}_e^{-1} R_e \underline{r}, \tag{2.22}$$

where $R_e$ is the restriction matrix that extracts nodal values of the residual vector that correspond to each overlapping domain, as indicated in fig. 2.9b. To improve the smoothing properties of the ASM, we introduce the diagonal weight matrix, $W_e$, which scales each nodal value by the inverse of the number of subdomains that share that node. Although it compromises symmetry, post-multiplication by $W_e$ was found to yield superior results to pre- and post-multiplication by $W_e^{\frac{1}{2}}$ [10, 11].




(a) (b) (c)

Figure 2.9: Figure 2.9a Approximation of deformed elements $\Omega_1$ and $\Omega_2$ as box-shaped, overlapping subdomains $\bar{\Omega}_1$ and $\bar{\Omega}_2$. Figure 2.9b overlapping subdomain $\tilde{\Omega}_e$, constructed by overlapping two nodes in each spatial dimension and applying a homogeneous Dirichlet boundary condition on $\partial\bar{\Omega}_e$. Figure 2.9c same as fig. 2.9b, but depicting copying the interior value of $\tilde{\Omega}_e$ to the overlapping region to communicate neighboring point values through `gslib`. This is further explained for ASM smoothing in alg. 2.10, RAS smoothing in alg. 2.11.

In a standard Galerkin ASM formulation, one would use $\bar{A}_e = R_e A R_e^T$, but such an approach would compromise the $O(p^3)$ storage complexity per element of the SE method. To construct fast inverses for $\bar{A}_e$, we approximate the deformed element as a simple box-like geometry, as demonstrated in fig. 2.9a. These boxes are then extended by a single degree-of-freedom in each spatial dimension to form overlapping subdomains with $\bar{p}^3 = (p+3)^3$ interior degrees-of-freedom in each domain. The approximate box domain enables the use of the fast diagonalization method (FDM) to solve for each of the overlapping subdomains, which can be applied in $O(Ep^4)$ time in $\mathbb{R}^3$. The extended-box Poisson operator is

$$\bar{A} = B_z \otimes B_y \otimes A_x + B_z \otimes A_y \otimes B_x + A_z \otimes B_y \otimes B_x, \tag{2.23}$$

where each $B_*, A_*$ represents the extended 1D mass-stiffness matrix pairs along the given dimension [2]. The FDM begins with a pre-processing step of solving a series of small, $\bar{p} \times \bar{p}$, generalized eigenvalue problems,

$$A_* \underline{s}_i = \lambda_i B_* \underline{s}_i \tag{2.24}$$

and defining $S_* = (\underline{s}_1 \dots \underline{s}_{\bar{p}})$ and $\Lambda_* = \text{diag}(\lambda_i)$, to yield the similarity transforms

$$S_*^T A_* S_* = \Lambda_*, \quad S_*^T B_* S_* = I. \tag{2.25}$$

From these, the inverse of the local Schwarz operator is

$$\bar{A}^{-1} = (S_z \otimes S_y \otimes S_x) D^{-1} (S_z^T \otimes S_y^T \otimes S_x^T), \tag{2.26}$$

where $D$ is a diagonal matrix defined as

$$D := I \otimes I \otimes \Lambda_x + I \otimes \Lambda_y \otimes I + \Lambda_z \otimes I \otimes I. \tag{2.27}$$

This process is repeated for each element, at each multigrid level save for the coarsest one. Note that the per-element storage is only $3\bar{p}^2$ for the $S_*$ matrices and $\bar{p}^3$ for $D$. At each multigrid level, the local subdomain solves are used as a smoother or the basis of a Chebyshev-accelerated polynomial smoother, see section 2.7 for more details. On the coarsest level ($p = 1$), however, BoomerAMG [43] as provided by `hypre` [17] is used to solve the system on the CPU with the same parameters described in section 2.4, except using Chebyshev smoothing. A single BoomerAMG V-cycle iteration is used in the coarse-grid solve.

Presently, we also consider a restrictive additive Schwarz (RAS) version of (2.22), wherein overlapping values are not added after the action of the local FDM solve, following [48],

$$S_{RAS} \underline{r} = \sum_{e=1}^{E} \tilde{R}_e^T \bar{A}_e^{-1} R_e \underline{r}. \tag{2.28}$$

RAS has the added benefit of reducing the amount of communication required in the smoother. Similar to ASM, RAS is *non-symmetric*. Attempts to symmetrize the operator tend to have a negative impact on the convergence rate [48]. The smoother $S_{ASM}$ or $S_{RAS}$ described here is then used to construct a polynomial smoother using Chebyshev polynomials of the 1st kind, as described in section 2.7, or 4th-kind, as described in section 4.1.

How are the smoothers described in eqs. (2.22) and (2.28) applied in parallel computing context, especially for heterogeneous architectures? Fortunately, the subdomain overlap operations are accomplished through a series of highly-scalable *gather-scatter* operations pro-

**Algorithm 2.10:** ASM smoother

**Data:** Initial solution, $\underline{x}$, Scratch buffers $\underline{x}_{ext}$, $\underline{r}_{ext}$
**Result:** Smooth solution, $\underline{r}$
**ParallelFor** $e = 1, \ldots, E$
  Copy element $e$ values from $\underline{x}$ into interior $\underline{x}_{ext}$
  Place element $e$ interior values of $\underline{x}_{ext}$ into $\underline{x}_{ext}$ `/* fig. 2.9c */`
**end**
`gslib`($\underline{x}_{ext}$) `/* perform on extended mesh */`
**ParallelFor** $e = 1, \ldots, E$
  Subtract $\underline{x}_{ext} \leftarrow \underline{x}_{ext} - \underline{x}$ for interior values
  Apply FDM for element $e$, $\underline{r}_{ext} \leftarrow (S_z \otimes S_y \otimes S_x) D^{-1} (S_z^T \otimes S_y^T \otimes S_x^T) \underline{x}_{ext}$
  Place element $e$ interior values of $\underline{r}_{ext}$ into $\underline{x}_{ext}$ `/* fig. 2.9c */`
**end**
`gslib`($\underline{r}_{ext}$) `/* perform on extended mesh */`
**ParallelFor** $e = 1, \ldots, E$
  Copy element $e$ exterior values of $\underline{r}_{ext}$ into interior $\underline{r}_{ext}$
  Place element $e$ interior values $\underline{r}_{ext}$ into $\underline{r}$
  $\underline{r} \leftarrow W_e \underline{r}$
**end**
`gslib`($\underline{r}$) `/* on regular mesh, ensure` $C^0$ `*/`

vided through the `gslib` library, as detailed in algs. 2.10 and 2.11. For more information, as well as kernel performance results for the FDM operation in eq. (2.26), see chapter 3. As shown in fig. 2.9c, the values in the overlapping region are communicated between neighboring elements by putting values that lie in the interior of $\tilde{\Omega}_e$ into the boundary of $\tilde{\Omega}_e$. After a single *gather-scatter* call and subtracting the values on the exterior of $\tilde{\Omega}_e$ by the values from the interior of $\tilde{\Omega}_e$, the value from the neighboring elements are exchanged. The corner vertices in the overlap region are ignored, as indicated in fig. 2.9b. The reasons for this are two-fold. First, overlapping corner values as in fig. 2.9b require a global element tensor-product structure, which is not possible given that we are developing solvers for *unstructured* problems. Second, this has the effect of reducing the number of elements involved in any *gather-scatter* exchange to the number of face neighbors, 4 in 2D and 6 in 3D, instead of $3^d - 1$ elements (8 in 2D, 26 in 3D). This greatly reduces the potential number of neighbors in the $QQ^T$ *gather-scatter* exchange, leading to a significant reduction in the number of messages and the amount of data communicated. When constructing the $S_x, S_y, S_z$ matrices in eq. (2.26), an element may lie on $\partial\Omega$, and therefore an overlapping point may lie outside of $\Omega$. In this scenario, the row corresponding to that point in the operator is set the zero. We note that the smoothers described in algs. 2.10 and 2.11 are employed as *smoothers* for a multigrid V-cycle which is embedded in a Krylov subspace method. Therefore, simplifying

assumptions such as the box-like approximation shown in fig. 2.9 are appropriate in this context.

**Algorithm 2.11:** RAS smoother

**Data:** Initial solution, $\underline{x}$, Scratch buffers $\underline{x}_{ext}$, $\underline{r}_{ext}$
**Result:** Smooth solution, $\underline{r}$
**ParallelFor** $e = 1, \ldots, E$
  Copy element $e$ values from $\underline{x}$ into interior $\underline{x}_{ext}$
  Place element $e$ interior values of $\underline{x}_{ext}$ into $\underline{x}_{ext}$ `/* fig. 2.9c */`
**end**
`gslib`($\underline{x}_{ext}$) `/* perform on extended mesh */`
**for** $e = 1, \ldots, E$ **do**
  Subtract $\underline{x}_{ext} \leftarrow \underline{x}_{ext} - \underline{x}$ for interior values
  Apply FDM for element $e$, $\underline{r}_{ext} \leftarrow (S_z \otimes S_y \otimes S_x) D^{-1} (S_z^T \otimes S_y^T \otimes S_x^T) \underline{x}_{ext}$
  Place element $e$ interior values $\underline{r}_{ext}$ into $\underline{r}$
  $\underline{r} \leftarrow W_e \underline{r}$ `/* inverse multiplicity */`
**end**
`gslib`($\underline{r}$) `/* on regular mesh, ensure` $C^0$ `*/`

The ASM and RAS smoothers in algs. 2.10 and 2.11 are used as smoothers in a multigrid V-cycle. The `Nek5000`-based hybrid Schwarz multigrid method described in section 2.5 utilizes a single ASM or RAS smoothing operation as a `pre-smooth` operation in an additive V-cycle, alg. 2.9. The `post-smooth` operation is omitted. As mentioned in section 2.5, the motivation behind this construction is that every matrix-vector product employed in the solver is *projected*—that is, every matrix-vector product performed in the algorithm is done by the outer KSP. These methods are denoted by the smoother, followed by the multigrid schedule. For example, the common $p = 7, 3, 1$ cycle with ASM smoothing is denoted ASM, (7,3,1). While this method is the only multigrid scheme available in `Nek5000`, we consider in this thesis more general multigrid schemes, including the multiplicative V-cycle from alg. 2.8. To further improve the smoothing properties of the ASM and RAS smoothers, we also consider their use as part of a polynomial smoother as described in section 2.7. In section 2.9, we compare the efficacy of the `Nek5000`-style algorithm compared with the other approaches described in this thesis.

## 2.7 CHEBYSHEV POLYNOMIAL SMOOTHERS

Consider the multigrid V-cycle approach as described in section 2.5. Given $k_j$ iterations of the smoothing iteration in eq. (2.18) for level $j$, is it possible to construct a better order

$k_j$ polynomial than $p_{k_j}(S_jA_j) = G_{k_j} = (I - \omega S_jA_j)^{k_j}$? Following [8, 31], we wish to solve:

$$\min_{p_k \in \mathbb{P}_k, p_k(0)=1} \max_{\lambda \in [\lambda_{min}, \lambda_{max}]} |p(t)|. \tag{2.29}$$

where $\mathbb{P}_k$ is the space of all polynomials with degree less than or equal to $k$. The solution to the *mini-max* problem in eq. (2.29) are the shifted and scaled Chebyshev polynomials of the 1st-kind

$$\hat{T_k}(\lambda) = \frac{1}{\sigma_k} T_k\left(\frac{\theta - \lambda}{\delta}\right) \text{ with } \sigma_k := T_k\left(\frac{\theta}{\delta}\right). \tag{2.30}$$

$T_k(\cdot)$ is the Chebyshev polynomial of the 1st-kind of order $k$; $\theta$ is the midpoint of the interval $[\lambda_{min}, \lambda_{max}]$,

$$\theta = \frac{\lambda_{min} + \lambda_{max}}{2}; \tag{2.31}$$

and $\delta$ is the mid-width of the interval,

$$\delta = \frac{\lambda_{max} - \lambda_{min}}{2}. \tag{2.32}$$

The Chebyshev polynomials of the 1st-kind enjoy a three-term recurrence relation that is used to derive alg. 2.12 [31]. The maximum eigenvalue of $S_jA_j$ is estimated through Arnoldi iteration, providing a high-quality estimate $\tilde{\lambda}$ for $\lambda_{max}$. As we later see, the quality of the Chebyshev smoother is highly sensitive to underestimation in the maximum eigenvalue. Therefore a small factor larger than unity is typically used, e.g., $\lambda_{max} = 1.1\tilde{\lambda}$. While $\lambda_{max}$ is taken to represent the largest eigenvalue of $S_jA_j$, how should $\lambda_{min}$ be chosen? $\lambda_{min}$ is chosen as a small factor of $\lambda_{max}$. Previous works considered factors such as 1/30 [8], 3/10 [49], 1/4 [7], and 1/6 [50].

To better understand the effect of the $\lambda_{min}$ and $\lambda_{max}$ parameters from alg. 2.12, a sensitivity analysis similar to that performed by Adams and coworkers [8] is conducted. For this, we consider the challenging Kershaw model problem, section 2.8.1, with $\varepsilon = 0.3$, $E = 24^3, p = 7$, and relative residual tolerance of $10^{-8}$. A third-order Chebyshev-accelerated ASM smoother is used in the $p = 7, 3, 1$ pMG V-cycle preconditioner.

The results of the sensitivity study are shown in table 2.2. The required iteration counts are seen to be sensitive to underestimation of $\lambda_{max}$, as also found by Adams and coworkers [8]. On the other hand, results are less sensitive to the minimum eigenvalue estimate, provided $\lambda_{min} > 0$, as this delimits between the low frequencies that are handled by the coarse grid correction and the high frequencies that are eliminated by the action of the smoother. Unless otherwise noted, we choose $\lambda_{min} = 0.1\tilde{\lambda}$, $\lambda_{max} = 1.1\tilde{\lambda}$ throughout this thesis.

**Algorithm 2.12:** Chebyshev smoother, 1st-kind

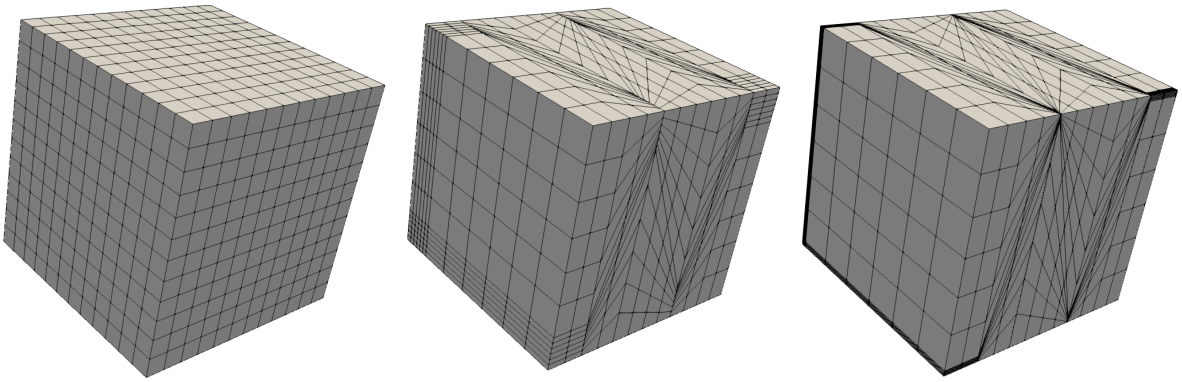
**Data:** Initial solution, $\underline{x}_0$; Chebyshev polynomial order, $k$
**Result:** Smooth solution, $\underline{x}_k$
$\theta = \frac{1}{2}(\lambda_{max} + \lambda_{min})$, $\delta = \frac{1}{2}(\lambda_{max} - \lambda_{min})$, $\sigma = \frac{\theta}{\delta}$, $\rho_0 = \frac{1}{\sigma}$
$\underline{x}_0 = \underline{x}$
$\underline{r}_0 = S(\underline{b} - A\underline{x}_0)$ `/* Omit` $A\underline{x}_0$ `if` $\underline{x}_0 = \underline{0}$ `*/`
$\underline{d}_0 = \frac{1}{\theta}\underline{r}_0$
**for** $i = 1, \ldots, k-1$ **do**
  $\underline{x}_i = \underline{x}_{i-1} + \underline{d}_{i-1}$
  $\underline{r}_i = \underline{r}_{i-1} - SA\underline{d}_{i-1}$, $\rho_i = \frac{1}{2\sigma - \rho_{i-1}}$
  $\underline{d}_i = \rho_i \rho_{i-1} \underline{d}_{i-1} + \frac{2\rho_i}{\delta}\underline{r}_i$
**end**
$\underline{x}_k = \underline{x}_{k-1} + \underline{d}_{k-1}$

Table 2.2: Iteration counts for the $(\lambda_{min}, \lambda_{max})$ sensitivity study. Omitted entries failed to converge in 1000 iterations.

| $\lambda_{min} \backslash \lambda_{max}$ | $0.9\tilde{\lambda}$ | $0.95\tilde{\lambda}$ | $1.0\tilde{\lambda}$ | $1.1\tilde{\lambda}$ | $1.2\tilde{\lambda}$ | $1.3\tilde{\lambda}$ |
|---|---|---|---|---|---|---|
| $0.0\tilde{\lambda}$ | - | - | - | - | - | - |
| $0.025\tilde{\lambda}$ | - | - | - | 110 | 64 | 48 |
| $0.05\tilde{\lambda}$ | - | - | - | 50 | 40 | 38 |
| $0.1\tilde{\lambda}$ | - | - | 124 | 40 | 38 | 38 |
| $0.2\tilde{\lambda}$ | 159 | 45 | 43 | 42 | 43 | 44 |
| $0.25\tilde{\lambda}$ | 47 | 45 | 44 | 44 | 45 | 46 |

## 2.8 CASES

We describe four model problems that are used to test the SE preconditioners. The first is a stand-alone Poisson solve, using variations of the Kershaw mesh. The others are modest-scale Navier-Stokes problems, where the pressure Poisson problem is solved over multiple time-steps. The problem sizes are listed in table 2.4 and range from relatively small ($n$=21M points) to moderately large ($n$=645M).[3]



(a) $\varepsilon = 1.0$ (b) $\varepsilon = 0.3$ (c) $\varepsilon = 0.05$

Figure 2.10: Kershaw, $E = 12^3, p = 1$.

### 2.8.1 Kershaw

The Kershaw family of meshes [51, 52] has been proposed as the basis for a high-order Poisson-solver benchmark by the Center for Efficient Exascale Discretization (CEED) within the DOE Exascale Computing Project (ECP). This family of meshes is parameterized by an anisotropy measure, $\varepsilon = \varepsilon_y = \varepsilon_z \in (0, 1]$, that determines the degree of deformation in the $y$ and $z$ directions. As $\varepsilon$ decreases, the mesh deformation and aspect ratio increase along with it. The Kershaw mesh is shown in fig. 2.10 for $\varepsilon = 1, 0.3, 0.05$. The domain is defined as $\Omega = [-1/2, 1/2]^3$ with Dirichlet boundary conditions on $\partial\Omega$. The right hand side for eq. (1.1) is set to

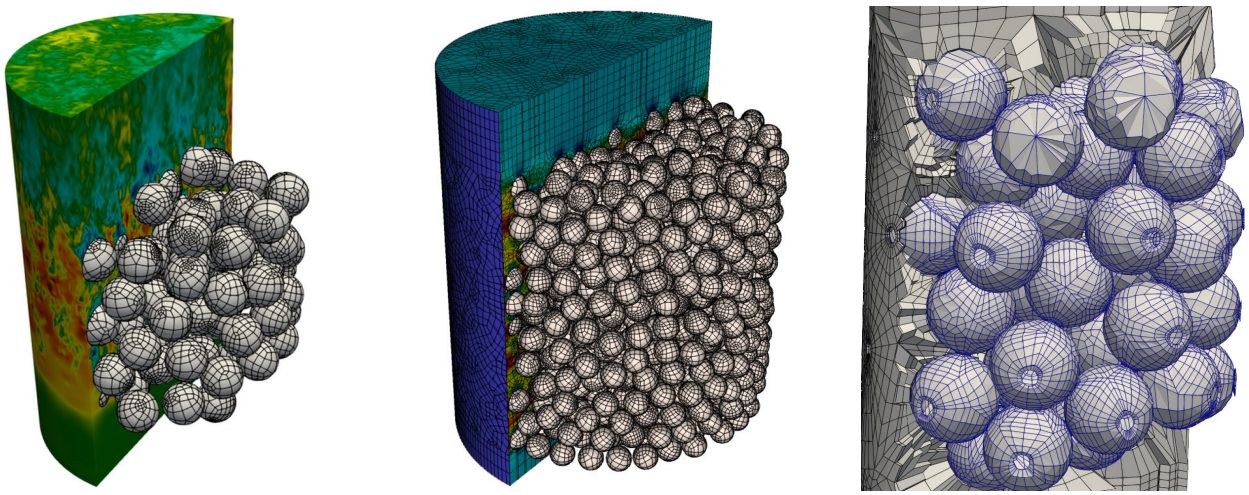
$$f(x, y, z) = 3\pi^2 \sin(\pi x) \sin(\pi y) \sin(\pi z) + g, \tag{2.33}$$

where $g(x, y, z)$ is a random, continuous vector vanishing on $\partial\Omega$. The linear solver terminates after reaching a relative reduction of $10^{-8}$. The solver used is GMRES(20) preconditioned with a single pMG V-cycle. Since this test case solves the Poisson equation, there is no time-stepper needed for the model problem. Mesh quality metrics are shown in table 2.3 for $E = 36^3, p = 7$.

### 2.8.2 Navier-Stokes Cases

For the pressure-Poisson tests, four flow cases are considered, as depicted in figs. 2.11 and 2.12. The three cases in fig. 2.11 corresponds to turbulent flow through a cylindrical packed-bed with 146, 1568, and 67 spherical pebbles. The 146 and 1568 pebble cases are from Lan and coworkers [53]. The 67 pebble case is constructed using an alternate Voronoi

[3]Larger cases for recent full-scale runs on Summit with $n$=51B are reported in [16].



(a) (b) (c)

Figure 2.11: Navier-Stokes pebble cases with: (a) 146, (b) 1568, and (c) 67 spheres.

cell approach, and includes chamfers [54, 55]. As such, the 67 pebble case is a more complex geometry, as indicated in table 2.3. The first two bed flows are at Reynolds number $Re_D = 5000$, based on sphere diameter, $D$, while the 67 pebble case is at Reynolds number $Re_D = 1460$. Time advancement is based on a two-stage 2nd-order characteristics time-stepper with CFL=4 ($\Delta t = 2 \times 10^{-3}$ $\Delta t = 5 \times 10^{-4}$, and $\Delta t = 5 \times 10^{-5}$ for the 146, 1568, and 67 pebble cases). An absolute pressure solver tolerance of $10^{-4}$ is used. A restart at $t = 10$, $t = 20$, and $t = 10$ convective time units is used for the 146, 1568, and 67 pebble cases, respectively, to provide an initially turbulent flow.

Table 2.3: Mesh quality metrics for cases from figs. 2.10 to 2.12.

| Case Name | Spacing (min/max) | Jac. (min/max/avg) | A.R. (min/max/avg) |
|---|---|---|---|
| Kershaw ($\varepsilon = 1$) | $1.78 \times 10^{-3}$ / $5.81 \times 10^{-3}$ | 1 / 1 / 1 | 1 / 1 / 1 |
| Kershaw ($\varepsilon = 0.3$) | $5.34 \times 10^{-4}$ / $3.5 \times 10^{-2}$ | 0.316 / 1 / 0.841 | 1.08 / 20.1 / 4.64 |
| Kershaw ($\varepsilon = 0.05$) | $8.91 \times 10^{-5}$ / $4.72 \times 10^{-2}$ | $1.86 \times 10^{-2}$ / 1 / 0.733 | 1.1 / 162 / 21.7 |
| 146 pebble | $1.01 \times 10^{-3}$ / .32 | $4.31 \times 10^{-2}$ / .977 / .419 | 1.07 / 56.9 / 7.14 |
| 1568 pebble | $2.21 \times 10^{-4}$ / .3 | $2.59 \times 10^{-2}$ / .99 / .371 | 1.12 / 108 / 12.6 |
| 67 pebble | $4.02 \times 10^{-5}$ / .145 | $5.97 \times 10^{-3}$ / .970 / .38 | 1.17 / 204 / 13.2 |
| Boeing speed bump | $8.34 \times 10^{-7}$ / .00299 | .996 / 1 / .999 | 6.25 / 255 / 28.1 |

The case shown in fig. 2.12 is a direct numerical simulation (DNS) of separated turbulent flow over a speed bump at $Re = 10^6$. This test case was designed by Boeing to provide a flow that exhibits separation. A DNS of the full 3D geometry, however, remains difficult [56]. Therefore, this smaller example proves a useful application for bench-marking solver performance. This case uses a 2nd-order time-stepper with CFL=0.8 ($\Delta t = 4.5 \times 10^{-6}$) and

an absolute pressure-solve tolerance of $10^{-5}$. A restart at $t = 5.6$ convective time units is used for the initial condition.

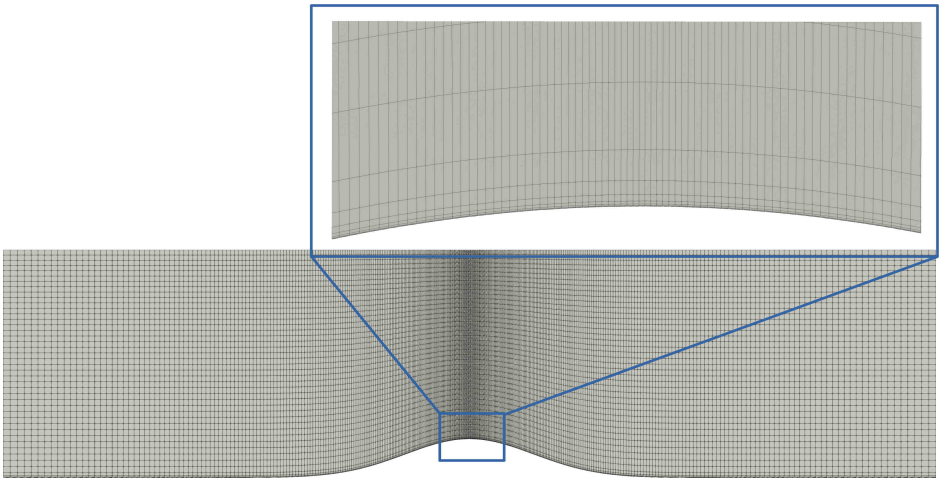


Figure 2.12: Direct numerical simulation (DNS) of Boeing speed bump.

In all cases, solver results are collected over 2000 time-steps. At each step, the solution is projected onto a space of up to $L = 10$ prior solution vectors to generate a high-quality initial guess, $\bar{\underline{u}}$. Projection is standard practice in `nekRS` as it can reduce the initial residual by orders of magnitude at the cost of just one or two matrix-vector products in $A$ per step [38]. Details regarding this projection scheme are discussed in section 2.3.

The perturbation solution, $\delta\underline{u} := \underline{u} - \bar{\underline{u}}$, is typically devoid of slowly evolving low wave-number content. Moreover, the initial residual is oftentimes sufficiently small that the solution converges in $k < 5$ iterations, such that the $O(k^2)$ overhead of GMRES is small. As discussed in section 2.2.4, GMRES is employed as the underlying ASM/RAS smoothers, presented in section 2.6, are non-symmetric. The additional expense incurred by using GMRES as opposed to CG, however, is further mitigated by the use of a projection scheme to generate a high-quality initial guess. Testing the preconditioners under these conditions ensures that the conclusions drawn are relevant to the application space.

Table 2.4: Problem discretization parameters.

| Case Name | $E$ | $p$ | $n$ |
|---|---|---|---|
| 146 pebble (fig. 2.11a) | 62K | 7 | 21M |
| 1568 pebble (fig. 2.11b) | 524K | 7 | 180M |
| 67 pebble (fig. 2.11c) | 122K | 7 | 42M |
| Speed bump (fig. 2.12) | 885K | 9 | 645M |

## 2.9 NUMERICAL RESULTS

Here we consider the solver performance results for the test cases in section 2.8, adapted from [25]. We assign a single MPI rank to each GPU and denote the number of ranks as $P$. All runs are on Summit. Each node on Summit consists of 42 IBM Power9 CPUs and 6 NVIDIA V100 GPUs. We use 6 GPUs per node unless $P < 6$. In the following, we denote a pMG preconditioner using $\eta$-order Chebyshev-accelerated $\xi$ smoother with a multigrid schedule of $\Pi$ as $1^{st}$-Cheb, $\xi(\eta,\eta),\Pi$. The `Nek5000`-style additive multigrid solvers employing ASM or RAS-based smoothing, as described in section 2.6, are similarly denoted as ASM,$\Pi$ and RAS,$\Pi$, respectively. Note that only a single pre-smoothing pass is considered for the latter solvers. For the moment, the Chebyshev polynomial smoothers considered are restricted to the first kind Chebyshev polynomial smoothers presented in section 2.7. The pre-/post-smoother use the same $\eta$-order Chebyshev smoother across all multigrid levels, except the coarsest. A wide range of preconditioning strategies is considered.

### 2.9.1 Kershaw

The Kershaw study (section 2.8.1) comprises six tests. For each of two studies, we consider the regular box case ($\varepsilon = 1.0$), a moderately skewed case ($\varepsilon = 0.3$), and a highly skewed case ($\varepsilon = 0.05$). The first study is a standard weak-scaling test, where $P$ and $E$ are increased, while the polynomial order is fixed at $p = 7$ and the number of gridpoints per GPU is set to $n/P = 2.67M$. The range of processors is $P$=6 to 384. The second study is a test of the influence of the polynomial order on conditioning, with $P = 24$ and $n/P = 2.88M$ fixed, while $p$ ranges from 3 to 10. Both cases use GMRES(20).

The results of the weak-scaling study are shown in fig. 2.13. For all values of $\varepsilon$, the iteration count exhibits a dependence on problem size, as seen in fig. 2.13a,d,g, especially in the highly skewed case ($\varepsilon = 0.05$). The time-per-solve also increases with $n$, in part due to the increase in iteration count, but also due to increased communication overhead as $P$ increases. This trend is not necessarily monotonic, as shown in fig. 2.13c,f. In the case of $\varepsilon = 0.3$ $1^{st}$-Cheb, RAS(3,3),(7,3,1), a minor fluctuation in the iteration count from the $P = 6$ to $P = 12$ case causes the greater than unity parallel efficiency. For $\varepsilon = 1.0$ $1^{st}$-Cheb, Jac(3,3),(7,5,3,1), the effects of system noise on the $P = 6$ run causes the greater than unity parallel efficiency for the $P = 12$ run.

Table 2.5 indicates the maximum number of of neighboring processors for the assembly ($QQ^T$) graph of $A$, which increases with $P$. In addition, the number of grid points per GPU ($n/P = 2.67M$) is relatively low. These two factors cause an increased sensitivity of

the problem to the additional communication overhead as the number of GPUs is initially increased. The number of neighbors, however, saturates at larger (i.e., production-level) processor counts.

Lastly, the relative preconditioner performance depends on $\varepsilon$. Figure 2.13b, e, h show that, for the easy $\varepsilon = 1.0$ case, a pMG scheme with a smoother that is cheap to apply is best, such as $1^{st}$-Cheb, RAS(2,2),(7,3,1), $1^{st}$-Cheb, Jac(3,3),(7,5,3,1), and ASM,(7,3,1). However, as $\varepsilon$ decreases, more robust pMG smoothers such as $1^{st}$-Cheb, RAS(3,3),(7,3,1) and SEMFEM result in lower time to solution. Once $\varepsilon = 0.05$ (fig. 2.13h), the problem is sufficiently challenging that SEMFEM overtakes the pMG based preconditioning schemes. This indicates that, in the highly skewed case in which the maximum element aspect ratio increases, the pMG preconditioner is not as effective as the SEMFEM preconditioner.

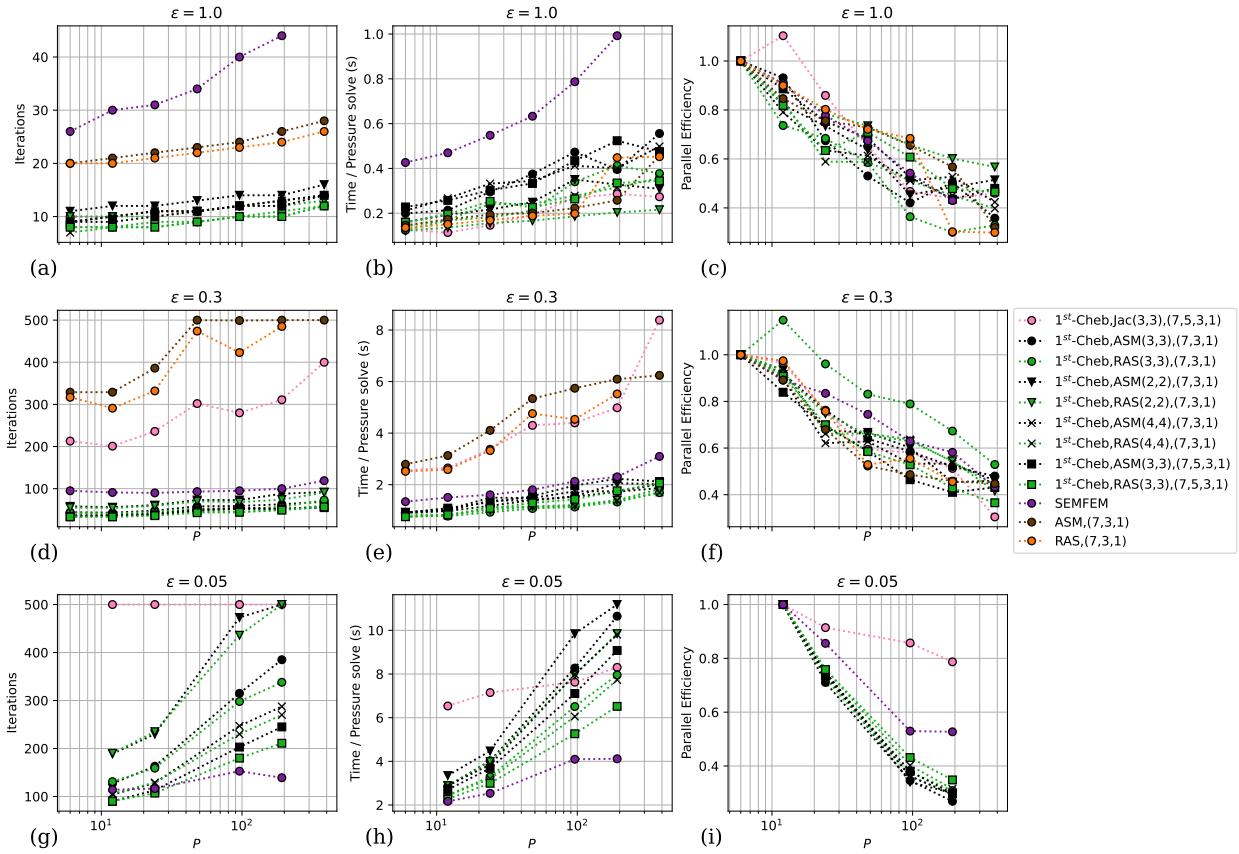



Figure 2.13: Kershaw weak scaling, GMRES(20). $n/P = 2.67M$.

The influence of the polynomial order is illustrated in fig. 2.14. For $\varepsilon = 1.0$, iteration counts are essentially $p$-independent, as seen in fig. 2.14a. For $\varepsilon = 0.3$, however, a slight upward trend in the iteration count is observed for SEMFEM and for the pMG preconditioners with ASM, RAS, and Chebyshev-Jacobi smoothing (fig. 2.14b). Similarly, $\varepsilon = 0.05$ exhibits a

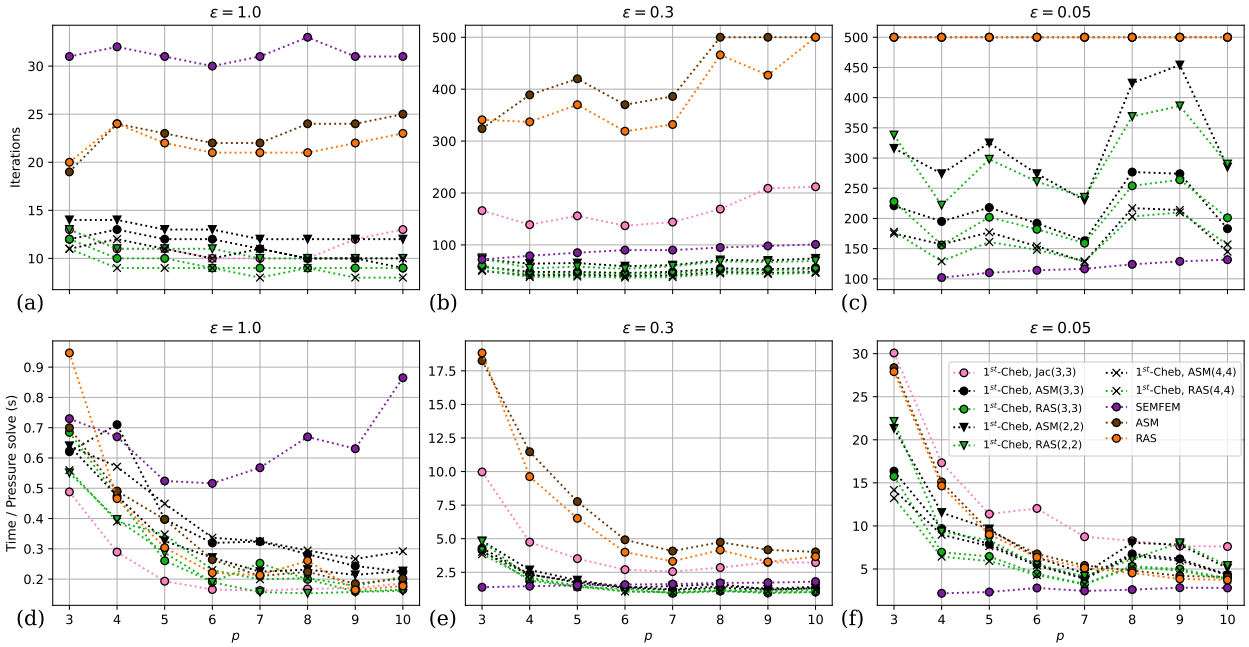



Figure 2.14: Kershaw order dependence, GMRES(20). $n/P = 2.88M$.

Table 2.5: Max neighbors, Kershaw.

| Nodes | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| Max Neighbors | 5 | 11 | 16 | 24 | 20 | 24 | 29 |

dependence between the iteration count and the polynomial order for all preconditioners (fig. 2.14c). Figure 2.14d, e and f demonstrate that the *time* per pressure solve is strongly dependent on the polynomial order. For SEMFEM, there is an increase in time as a result of increased overhead for the underlying AMG solver. For the pMG-based methods, higher orders are generally faster. This performance gain can be attributed to surface-to-volume effects in the evaluation of the operators and smoothers. Application of $A^e$ and (smoother) $S^e$ is highly vectorizable, whereas application of $QQ^T$ (assembly) involves a significant amount of indirect addressing. The discrete surface to volume ratio for a spectral element of order $p = 7$ is $296/512 \approx 60\%$. For lower $p$ this value is larger. This situation is exacerbated in the case of Schwarz-based smoothers, where the overlap contributes substantially to the work and communication for the small-element (i.e., low-$p$) cases. In addition, these results were collected with an older version of `nekRS` [4], which used a single element per thread block, thereby limiting the amount of work available for a streaming multiprocessor for relatively small polynomial orders. This issue, however, is addressed in newer versions of `nekRS` through the run-time kernel tuning strategies discussed in section 3.2.

### 2.9.2 Navier-Stokes

We consider scalability of `nekRS` for the cases of figs. 2.11 and 2.12. All simulations except one use GMRES(15) with an initial guess generated by $A$-conjugate projection onto $L = 10$ prior solutions [38]. Due to memory constraints, the 1568-pebble case with $P = 24$ uses GMRES(10) with only $L = 5$ solution-projection vectors. For each case, two pMG schedules are considered: $(7, 5, 3, 1)$ and $(7, 3, 1)$ for $p = 7$; and $(9, 7, 5, 1)$ and $(9, 5, 1)$ for $p = 9$. Other parameters, such as the Chebyshev order and the number of coarse grid BoomerAMG V-cycles are also varied. Results are shown in figs. 2.15 and 2.16b. The plots relate the effective work rate per node, measured as the gridpoints $n$ (as shown in table 2.4) solved per second per node, to the time-to-solution. The y-axis notes the drop in the relative work rate, which corresponds to a lower parallel efficiency, as the strong scale limit is reached, while the x-axis denotes the time-to-solution. Each node consists of 6 GPUs, hence $P = 6 \times$ nodes.

In all the performance tests conducted, the pMG preconditioner with Chebyshev-Jacobi smoothing is outperformed by the other preconditioners, whether using one or two V-cycle iterations in the AMG coarse-grid solve. For each case, the fastest preconditioner scheme varies. In the 146 pebble case (fig. 2.15a), using $1^{st}$-Cheb, RAS(3,3),(7,5,3,1) yields the smallest time per pressure solve. However, in the 1568 pebble case (fig. 2.15b), SEMFEM is a moderate improvement over the second best preconditioner, $1^{st}$-Cheb, ASM(3,3),(7,5,3,1). pMG with a $(9, 5, 1)$ schedule and Chebyshev-RAS (of any order) yield the best scalability and lowest time per pressure solve for the Boeing speed bump case (fig. 2.16b). The Chebyshev-accelerated Schwarz schemes are not always the fastest, however. For the 67 pebble case (fig. 2.15c), $1^{st}$-Cheb, Jac(3,3),(7,5,3,1) is comparable to $1^{st}$-Cheb, RAS(3,3),(7,5,3,1) and are the two fastest pMG based preconditioners. However, SEMFEM is significantly faster than the other preconditioners for this case.

Also considered is a two-level approach wherein SEMFEM is used as a non-Galerkin approximation to the coarse grid operator. Algorithm 6.1 describes this approach, wherein the SEMFEM operator is used as the coarse grid operator at a different polynomial order. A complete description of this approach, however, is deferred to chapter 6. For $p = 7$, a $(7, 6)$ schedule with 3rd order Chebyshev-accelerated ASM smoothing on the $p = 7$ level and SEMFEM solver on the $p = 6$ level, denoted as $1^{st}$-Cheb, ASM(3,3),(7,6) + SEMFEM, is used. Similarly, $1^{st}$-Cheb, ASM(3,3),(7,5) + SEMFEM and $1^{st}$-Cheb, ASM(3,3),(7,3) + SEMFEM are considered. For $p = 9$, a $(9, 8)$, $(9, 7)$, $(9, 5)$, and $(9, 3)$ two-level approach is considered, denoted as $1^{st}$-Cheb, ASM(3,3),(9,8) + SEMFEM, $1^{st}$-Cheb, ASM(3,3),(9,7) + SEMFEM, $1^{st}$-Cheb, ASM(3,3),(9,5) + SEMFEM, and $1^{st}$-Cheb, ASM(3,3),(9,3) + SEMFEM, respectively. In the pebble cases shown in fig. 2.15a,c, this two-level approach performs

somewhere between the SEMFEM and $1^{st}$-Cheb, ASM(3,3),(7,3,1) preconditioners. In the Boeing speed bump case, however, fig. 2.16b demonstrates that this approach is not as performant as either the SEMFEM or $1^{st}$-Cheb, ASM(3,3),(9,5,1) preconditioners. Theoretical analysis of this two-level approach based on two-grid perturbed coarse-grid operators from [57] is deferred to section 6.2, where it is shown that using this two-level approach with a polynomial order less than the fine level worsens the condition number compared to the spectral equivalence discussed in section 2.4. This idea, however, is further explored and improved in chapter 6.

Solver performance degrades whenever $n/P$ is sufficiently small, regardless of the solver considered. For the various preconditioners considered, at 2 nodes ($n/P = 1.75M$), the strong-scale limit of 80% efficiency is far surpassed in the 146 pebble case. This leaves the effective strong scale limit at 1-2 nodes ($n/P = 3.5$ to $1.75M$). For the 1568 pebble case, 12 nodes ($n/P = 2.5M$) yields a parallel efficiency around 60-70%, depending on the specific solver. The 67 pebble cases reaches 70% efficiency on 3 nodes ($n/P = 2.3M$). The parallel efficiency for the Boeing speed bump case for the fastest time-to-solution preconditioners drops below 70% when using more than 48 nodes ($n/P = 2.24M$).

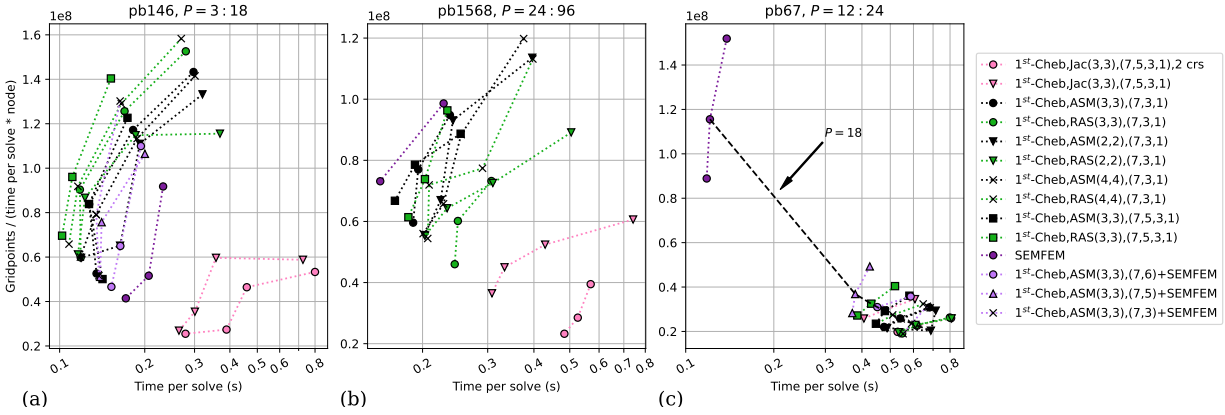
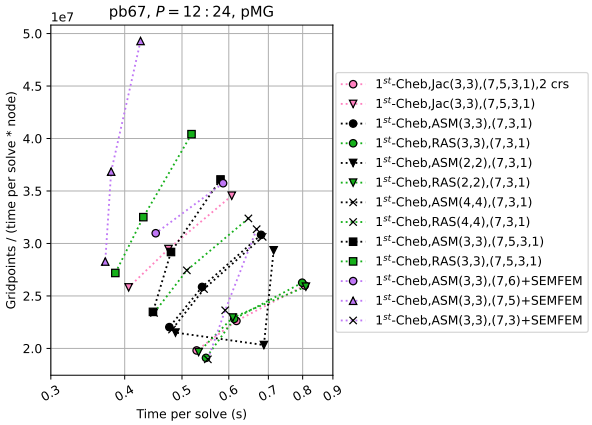
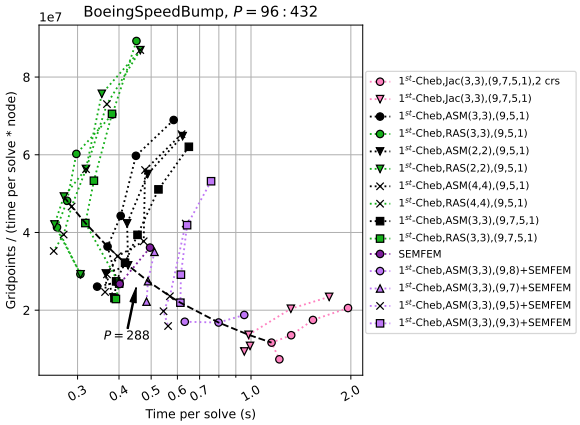



Figure 2.15: Strong scaling results on Summit for the Navier-Stokes cases of fig. 2.11. Iso-processor count line illustrated in (c). A user running on a specified number of processors should use the lowest time-to-solution preconditioner along this line.

While the effect of mesh quality metrics, such as the max aspect ratio and scaled Jacobian, on solver convergence has been studied by Mittal and coworkers [58], predicting the optimal preconditioner settings from mesh quality metrics is not obvious. While the maximum aspect ratio is an important metric for mesh quality, it alone cannot explain the apparent poor performance of the pMG preconditioners in the 67 pebble case, fig. 2.15c. Consider, for example, that the Boeing speed bump case has a larger maximum aspect ratio (255) than the 67 pebble case (204), but does not exhibit this poor performance in the pMG preconditioners.




(a) Same as fig. 2.15c, pMG preconditioners only.




(b) Strong scaling results on Summit for the Navier-Stokes case, fig. 2.12.

Figure 2.16: fig. 2.15: Strong scaling results for 67 pebble case, limited to $p$-geometric multigrid methods. fig. 2.16b: Strong scaling results for the Boeing speed bump case.

The minimum scaled Jacobian, however, is at least an order of magnitude smaller in the 67 pebble case ($5.97 \times 10^{-3}$) as compared to the other cases (e.g., .996 for the Boeing speed bump case, $4.31 \times 10^{-2}$ and $2.59 \times 10^{-2}$ for the 146, 1568 pebble cases, respectively). This may, in turn, explain why the pMG preconditioners in the 67 pebble case were significantly sub-optimal compared to the SEMFEM preconditioner. This demonstrates, however, that a user cannot simply rely on, e.g., the maximum aspect ratio when deciding whether or not to use SEMFEM as a preconditioner. This inability to correctly identify preconditioner settings based on mesh quality metrics alone motivates the introduction of an auto-tuner to choose preconditioner parameters during run-time, which is the subject of section 3.1.

Table 2.6: Strong scaling limits and fastest solver for the Navier-Stokes cases from section 2.8.2. The time-to-solution for the best solver $T_s$ is reported. Speedup over the time-to-solution for the default solver, $T_D$, is reported in the last column. For $p = 7$, the default solver is $1^{st}$-Cheb, ASM(3,3),(7,3,1). For $p = 9$, the default solver is $1^{st}$-Cheb, ASM(3,3),(9,3,1). For $E$, $p$, and $n$, refer to table 2.4.

| Case Name | $P$ | $n/P$ | Fastest Solver | $T_s$ | $\frac{T_D}{T_s}$ |
|---|---|---|---|---|---|
| 146 pebble | 6 | 3.5M | $1^{st}$-Cheb, RAS(3,3),(7,5,3,1) | 0.15 | 1.20 |
| 1568 pebble | 72 | 2.5M | SEMFEM | 0.18 | 1.08 |
| 67 pebble | 18 | 2.3M | SEMFEM | 0.12 | 4.51 |
| Speed bump | 288 | 2.2M | $1^{st}$-Cheb, RAS(3,3),(9,5,1) | 0.28 | 1.32 |

The fastest solver configuration and strong scaling limits for the Navier-Stokes cases are summarized in table 2.6. Through carefully designing the preconditioner, we achieve modest

speedups of 8%, 20%, and 32% for the 1568 pebble, 146 pebble, and Boeing speed bump cases, respectively. This is with respect to a performant $p$-multigrid default preconditioner based on Chebyshev-accelerated ASM smoothing. For the 67 pebble case, however, we observe that SEMFEM preconditioning is required to achieve good solver performance. As a result, there is no single preconditioner configuration that runs well across all cases. In section 3.1, we propose a *run-time* tuning approach to help mitigate this issue.

Table 2.6 establishes that the strong scalability limit remains relatively constant at $n/P \sim 2.5M$ despite the overall problem size. The scalability limits identified in table 2.6 are utilized in the remainder of this work. The "rule-of-thumb" for the Navier-Stokes cases considered herein is that strong scaling below $n/P \sim 2M$ for V100-based systems is not recommended. For newer, faster GPU architectures, this limit is increased due to the increased discrepancy between GPU compute and communication performance. Later in this thesis, section 3.2 establishes single GPU performance profiles of the individual kernels required for the preconditioners considered in this chapter. These results are relevant both to the newer A100 GPUs, as well as AMD's MI250X GPUs. We observe that achieving good *single GPU* performance requires *at least* $n/P \sim 1M$ points. This is further exacerbated once communication, both *on-node* and *off-node* is considered. This issue is further discussed in section 3.3.

## Chapter 3: Programming Considerations for Heterogeneous Architectures

As demonstrated in section 2.9, the methods discussed in chapter 2 exhibit good performance on the V100 GPUs used in the Summit supercomputer. Several prerequisites, however, are required to achieve this performance. First, the overall time-to-solution is highly dependent on the solver configuration, which tends to vary on a case-by-case basis. Second, the performance of the preconditioners depends both on highly performant GPU kernels and communication operations. The latter is achieved through highly scalable non-blocking point-to-point communication associated with performing *gather-scatter* operations on the GPU. The former, however, is the subject of this chapter. In this chapter, we will discuss strategies to achieve high-performance on GPU architectures, both from the prospective of a single GPU with the kernel tuning strategies discussed in section 3.2, as well multiple GPUs *at scale* through the tuning strategies discussed in sections 3.1 and 3.3.

This chapter is organized as follows. As observed in section 2.9, the optimal solver configuration depends highly on the problem at hand. Section 3.1 seeks to mitigate this issue by providing a description of a simple, yet effective, *online* tuning strategy that can be used to select optimal or nearly-optimal solver configurations during the course of a simulation. Section 3.2 then describes the two most important GPU kernels for the construction of fast, high-order GPU Poisson solvers: the matrix-vector product kernel and the fast diagonalization method (FDM) kernel employed in the Schwarz-based smoothers from section 2.6. As the optimal kernel configuration depends on the precision, polynomial order, and specific GPU architecture, section 3.2 describes an *online* tuning strategy to select highly performant kernels. Finally, the costs associated with the communication required in the pressure Poisson solves are quantified in section 3.3. The authors conduct a weak-scaling study to quantify the communication costs associated with the common operators described in the preconditioning methods from chapter 2.

### 3.1 PERFORMANCE PORTABILITY THROUGH RUN-TIME PRECONDITIONER SELECTION

The results of section 2.9 provides a small window into the varieties of performance behavior encountered in actual production cases, which span a large range of problem sizes, domain typologies, mesh qualities. Moreover, production simulations are solved across a range of architectures having varying on-node and network performance, interconnect typologies, and processor counts. One frequently encounters situations where certain communication pat-

terns might be slower under a particular MPI version on one platform versus another. Unless a developer has access to that platform, it is difficult to measure and quantify the communication overhead. Processor count alone can be a major factor in preconditioner selection: large processor counts have relatively high coarse-grid solve costs that can be mitigated by doing more smoothing at the fine and intermediate levels. How much more is the open question. The enormity of parameter space, particularly "in the field" (i.e., users working on unknown platforms) limits the effectiveness of standard complexity analysis in selecting the optimal preconditioner for a given user's application.

From the user's perspective, there is only one application (at a time, typically), and one processor count of interest. Being able to provide optimized performance—tuned to the application at hand, which includes the processor count—is thus of paramount importance. Auto-tuning provides an effective way to deliver this performance. Auto-tuning of preconditioners has been considered in early work by Imagery *et al.* [59] and more recently by Yamada *et al.* [60] and by Brown *et al.* [61]. The latter work couples robust optimization routines with local Fourier analysis to provide the objective function for the optimizer. Local Fourier analysis (LFA) is a powerful tool for analyzing the convergence properties of multigrid methods. A brief introduction to LFA for $p$-multigrid methods is deferred to section 4.3.

Table 3.1: Solver parameter space considered in the auto-tuner. A pMG preconditioner using an $\eta$-order Chebyshev-accelerated $\xi$ smoother with a multigrid schedule of $\Pi$ is denoted as Cheby-$\xi(\eta,\eta)$,$\Pi$.

| | Parameters |
|---|---|
| Solver | preconditioned GMRES |
| Preconditioner | pMG, SEMFEM |
| pMG | $p = 7$:<br>$1^{st}$-Cheb, ASM(3, 3),(7,3,1)<br>$1^{st}$-Cheb, RAS(3, 3), (7,3,1)<br>$1^{st}$-Cheb, Jac(3, 3), (7,5,3,1)<br>$p = 9$:<br>$1^{st}$-Cheb, ASM(3, 3),(9,5,1)<br>$1^{st}$-Cheb, RAS(3, 3), (9,5,1)<br>$1^{st}$-Cheb, Jac(3, 3), (9,7,5,1) |
| Coarse grid | single boomerAMG V-cycle |
| SEMFEM | single AmgX V-cycle |

In large-scale fluid mechanics applications, auto-tuning overhead is typically amortized over $10^4$–$10^5$ time-steps (i.e., pressure solves) *per run* (and more, over an entire simulation campaign). Moreover, auto-tuning is of particular importance for problems at large processor counts because these cases often have long queue times, which preclude making multiple job

submissions in order to tweak parameter settings. Failure to optimize, however, can result in significant opportunity costs. For example, in [16] the authors realized a factor of 2.8 speedup in time-per-step for a 352,000-pebble-bed simulation ($n$=51B gridpoints) through a sequence of tuning steps. Even if there were 100 configurations in the preconditioner parameter space, an auto-tuner could visit each of these in succession 5 times each within the first 500 steps and have expended only a modest increment in overhead when compared to the cost of submitting many jobs (for tuning) or to the cost of 10,000 steps for a production run.

**Algorithm 3.1:** Online Solver Tuner

**Input:** Solver configuration space, $\mathcal{S}'$, starting step to tune, $i_S$, initial solver configuration, $s_0$, objective function to minimize, $\phi$, number of samples, $N$, number of simulation time-steps, $i_f$
**Output:** Optimal solver configuration, $s^* \in \mathcal{S}'$
Construct stack $S$ containing $\forall v \in \mathcal{S}'$
$s \leftarrow s_0, \phi^* \leftarrow \infty,$ `converged` $\leftarrow$ `false`
**for** $i = 0, \ldots, i_f$ **do**
  **if** $i > i_S$ *and* $(i - i_S)\%N = 0$ *and not* `converged` **then**
    $(T_s, \rho_s, \ldots) \leftarrow$ statistics of $s$
    **if** $\phi(T_s, \rho_s, \ldots) < \phi^*$ **then**
      $s^* \leftarrow s, \phi^* \leftarrow \phi(T_s, \rho_s, \ldots)$
    **end**
    **if** $S$ *is empty* **then**
      $s \leftarrow s^*$ `/* found best solver, save result */`
      `converged` $\leftarrow$ `true`
    **else**
      $s \leftarrow S.pop()$ `/* try next solver */`
  **end**
  **if** $i = i_S$ **then**
    Enable sampling
  **end**
  Solve $A\underline{x}_i = \underline{b}_i$ using $s$
**end**
**return** $s^*$

As a preliminary step, the authors consider constructing a small subset of the true search space (which could include, e.g., other cycles, schedules, or smoothers) for use in a nascent auto-tuner. Algorithm 3.1 describes the auto-tuner used in this work. Most notably, the auto-tuner is designed to be used *online* during a run. This enables the simulation to continue to progress while the auto-tuner performs an exhaustive search over a limited subset of the full parameter space. Algorithm 3.1, moreover, requires several inputs. These include: the number of evaluation phases, $N$, to perform prior to noting the performance of a solver; the

starting step $i_S$ at which to begin the tuning procedure; and the parameter space $\mathcal{S}' \subset \mathcal{S}$ over which to search. While the latter is currently chosen as the solvers identified in table 3.1, the $N$ and $i_S$ parameters are exposed to the user, with reasonable defaults being $N = 5$ and $i_S = 100$. In addition, after each solver is evaluated under this tuning procedure, the best solver configuration $s^* \in \mathcal{S}'$ is noted. This has the additional benefit of providing a reasonably performant solver configuration, which the user can use on subsequent runs for the same case, e.g., for restarting a simulation to evolve the state to a later time.

Across all cases, with exception to the 67 pebble case, pMG preconditioning with Chebyshev-accelerated ASM or RAS smoothing is the fastest solver or is comparable to SEMFEM. The choice of Chebyshev order and multigrid schedule, moreover, contributes only a modest $\approx$ 10-20% improvement to the overall time-to-solution in most cases, all else being equal. This makes the default $1^{st}$-Cheb, ASM(3,3),(7,3,1) or $1^{st}$-Cheb, ASM(3,3),(9,5,1) preconditioner reasonably performant. However, in order to avoid the situation encountered in the 67 pebble case, SEMFEM is added to the considered search space. The parameters in table 3.1 are chosen as they include optimal or near-optimal preconditioner settings for the results in figs. 2.15 and 2.16b, while still restricting the search space to something that is amenable to exhaustive search. The authors note, however, that this parameter space may not reflect the various factors affecting the performance of the preconditioners at especially large $P$ or on different machine architectures.

Algorithm 3.1 details the online auto-tuner considered in this thesis. During the simulation, our simple auto-tuner performs an exhaustive search over the small parameter space identified in table 3.1. This parameter space, denoted as $\mathcal{S}'$, is a small heuristic subset of the true search space $\mathcal{S}$, $\mathcal{S}' \subset \mathcal{S}$. The cardinality of $\mathcal{S}'$ identified in table 3.1 is 4, so the space is amenable to exhaustive search. The authors note that the parameter space in $\mathcal{S}'$ can be expanded to include far more parameters, so long as the resultant space remains amenable to exhaustive search. As each candidate solver in $\mathcal{S}'$ is evaluated *online* during a simulation, the simulation is able to make progress as outlined in alg. 3.1. Therefore, the expense of searching over a somewhat larger parameter space is amortized over the course of the simulation.

In addition to the importance of identifying a heuristic search space $\mathcal{S}' \subset \mathcal{S}$ that is small enough to be amenable to exhaustive search while still being *rich* enough to include the optimal or near-optimal preconditioner, the authors note that the objective function, $\phi$ in alg. 3.1, must be carefully chosen. While the time-to-solution seems like a natural choice for the objective function, the authors note that utilizing this parameter alone can lead to sub-optimal results when combined with the high-quality initial guesses provided through the solution projection method described in section 2.3. As noted in the PCG error bound

eq. (2.3) in section 2.2.1, the convergence rate of the KSP solver is a function of both the condition number of the preconditioned operator, $\kappa(M^{-1}A)$, as well as the proximity of the initial guess to the true solution. The former is a function of the particular solver configuration, $s$, while the latter has no dependence on the solver at-hand. During the evaluation phase in alg. 3.1, a candidate solver configuration, $s$, is evaluated with the particular right-hand side and initial guess pairs for the given time-step. However, if the solution projection space is not sufficiently rich, the candidate configuration may be discounted as sub-optimal when in fact it is not. At the same time, other metrics, such as the solver convergence rate, do not fully take into account the expense of the preconditioner.

To address the issue highlighted above, consider that we are trying to find the solver configuration, $s^* \in \mathcal{S}'$, that minimizes the time-to-solution, $T_s = T_s(s)$. Let the iteration count be denoted by $k$ and define the average convergence rate for $s$ as

$$\rho_s = \exp\left(\frac{1}{k}\log\frac{\|\underline{r}_k\|_2}{\|\underline{r}_0\|_2}\right). \tag{3.1}$$

The following relationship holds:

$$\|\underline{r}_k\|_2 = \|\underline{r}_0\|_2\rho_s^k. \tag{3.2}$$

Let the solver converge when the residual norm is $\tau$, $\|\underline{r}_k\|_2 = \tau$. Then, from eq. (3.2), the number of iterations required to converge is

$$k = \log\frac{\tau}{\|\underline{r}_0\|_2}\cdot\frac{1}{\log\rho_s}. \tag{3.3}$$

Minimizing the time-to-solution, we observe that:

$$\begin{aligned}
s^* &= \arg\min_{s\in\mathcal{S}'} T_s \\
&= \arg\min_{s\in\mathcal{S}'} \underbrace{k}_{\text{eq. (3.3)}}\cdot\left(\frac{T_s}{k}\right) \\
&= \arg\min_{s\in\mathcal{S}'} \underbrace{\log\frac{\tau}{\|\underline{r}_0\|_2}}_{\text{const w.r.t. } s}\cdot\frac{1}{\log\rho_s}\cdot\left(\frac{T_s}{k}\right) \\
&= \arg\min_{s\in\mathcal{S}'} \frac{1}{\log\rho_s}\cdot\left(\frac{T_s}{k}\right).
\end{aligned} \tag{3.4}$$

Choosing the objective function as

$$\phi(s) = \frac{1}{\log \rho_s} \cdot \left( \frac{T_s}{k} \right), \tag{3.5}$$

therefore, provides a metric that balances the average solver convergence rate, $\rho_s$, and the average time per iteration, $T_s/k$, of the solver. Further, eq. (3.5) provides a proxy for the time-to-solution that does not depend on the right-hand side/initial guess dependent $\|\underline{r}_0\|_2$ term that is implicitly present in the time-to-solution metric.

The authors note that the choice of objective function in eq. (3.5) is important. For example, using the time-to-solution as the metric without regard for the quality of the initial guess during a simulation can lead the auto-tuner to select solver configurations that are highly sub-optimal. For example, a tentative solver configuration $s \in \mathcal{S}'$ may be evaluated during a handful of simulation steps with a *rich* projection space. The solver then converges in a small number of iterations, even with a relatively poor choice of preconditioner. For example, using the time-to-solution as the only metric, the auto-tuner *would* have chosen $1^{st}$-Cheb, Jacobi(3,3), (7,5,3,1) with a single AMG V-cycle approximating the coarse-grid solve as the preconditioner for the 146 pebble case. This is due to the fact that the evaluation phase for the $1^{st}$-Cheb, Jacobi(3,3), (7,5,3,1) configuration happens to coincide within a region of the simulation where the solution projection space is sufficiently rich, yielding low iteration counts. Visual inspection of fig. 2.11a, however, shows that this is a poor choice over the course of the entire simulation. However, by using the carefully crafted objective function in eq. (3.5), the $1^{st}$-Cheb, RAS(3,3),(7,3,1) preconditioner is correctly chosen.

While the heuristic search space, $\mathcal{S}'$ is limited compared to the true search space $\mathcal{S}$, the resultant preconditioners selected are effective. Applying the online tuner from alg. 3.1 with the heuristic space $\mathcal{S}'$ identified in table 3.1 to the strong scaling studies conducted in section 2.9.2, we observe that the tuner is able to identify optimal or nearly optimal solver configurations. For example, in the 146 pebble case (fig. 2.15a), $1^{st}$-Cheb, RAS(3, 3),(7,3,1) is identified as the preconditioner on each of the processor counts, which was comparable in performance to the best preconditioner. The auto-tuner chose the optimal SEMFEM for the 1568 pebble case (fig. 2.15b). SEMFEM was identified at the preconditioner for the 67 pebble case on all processor counts, which fig. 2.15c confirms. In the Boeing speed bump case (fig. 2.16b), the auto-tuner chose $1^{st}$-Cheb, RAS(3, 3),(9,5,1) across all processor counts, which was either the optimal or near-optimal preconditioner for the problem. These results are summarized in table 3.2.

Note that the auto-tuner selects the fastest method at a *fixed-processor count* (i.e., whatever the user has selected). Consider, for example, the 67 pebble case on $P = 18$ GPUs

Table 3.2: Outcome of running alg. 3.1 on the Navier-Stokes cases from section 2.8 with the parameter space identified in table 3.1.

| Case Name | Solver |
|---|---|
| 146 pebble (fig. 2.11a) | $1^{st}$-Cheb, RAS(3, 3),(7,3,1) |
| 1568 pebble (fig. 2.11b) | SEMFEM |
| 67 pebble (fig. 2.11c) | SEMFEM |
| Speed bump (fig. 2.12) | $1^{st}$-Cheb, RAS(3, 3),(9,5,1) |

(dashed black line, fig. 2.15c). The goal of the auto-tuner is to select the preconditioner with lowest time-to-solution along the user-specified iso-processor count line. However, in the results discussed above, there was no change with respect to a change in the processor count. We note that our principal objective is not to squeeze out a few percent over a raft of good choices, but rather to ensure that a case does not run with an unfortunate set of parameters for which the performance is significantly substandard. This primitive auto-tuning technique proves effective at preventing the selection of highly sub-optimal preconditioners. We have implemented this for production use and will continue to update and refine the strategy to include the novel methods considered in chapters 4 to 6.

## 3.2 PERFORMANCE PORTABLE SOLVER KERNELS

Two crucial GPU kernels required in efficiently solving the high-order Poisson problem are the matrix-vector product operator and the fast diagonalization method (FDM) operator from eqs. (2.26) and (2.27) as described in section 2.6. The matrix-vector product kernel is required both in the Krylov solver, as well as the construction of polynomial smoothers as discussed in section 2.7 and the multigrid residual re-evaluation in the standard V-cycle algorithm, see alg. 2.8. The FDM kernel, however, is only required in the construction of the polynomial smoothers. Both of these kernels, moreover, encompass the majority of solver operations utilizing the GPU. Due to their importance, tuning the matrix-vector product kernel in section 3.2.1 and the FDM kernel in section 3.2.2 is the primary focus of this section.

The kernels, along with the rest of the GPU kernels required in `nekRS`, are implemented in the Open Concurrent Compute Abstraction (`OCCA`) library [62]. `OCCA` is a performance portable library for writing GPU kernels that supports multiple back-ends, including CUDA, HIP, OpenCL, and OpenMP. One main feature of `OCCA` is its reliance on just-in-time (JIT) compilation. This allows for providing the compiler rich information, including the quadrature order and constant compile-time parameters such as the reference-element derivative

matrices. In addition, `OCCA` allows for easy run-time kernel selection, which is a prominent technique in obtaining good performance portable kernels in sections 3.2.1 and 3.2.2.

While the matrix-vector product associated with the Krylov solver must be performed in double precision for accuracy, the matrix-vector product operations associated with the preconditioner are performed in single precision to increase the overall throughput of the solver. This mixed-precision approach offers better solver performance without impacting accuracy or convergence. To fully exploit this mixed-precision approach, however, the matrix-vector product kernel must be performant in both precisions. The FDM kernel, on the other hand, is only employed as part of the preconditioner, and therefore must only target single precision.

The use of run-time kernel selection techniques discussed in this section is not new. On CPU-architectures, for example, the use of run-time code generation in `LIBXSMM` is considered by Heinecke and coworkers to generate highly optimized, small matrix-matrix product routines [63]. In [64], Hess and coworkers apply similar strategies to accelerate `Nek5000` on CPU-architectures. The optimization of sparse matrix-vector product kernels on GPU-architectures is explored by Baskaran and Bordawekar in [65] and more recently in a pre-print by Ashoury and coworkers [66]. The use of run-time code generation through user-guided code transformation statements is a prominent feature in `Loo.py` [67]. In [68], Klöckner and coworkers consider the use of `Loo.py` to generate highly-optimized kernels for high-order finite element operators. The approach considered in this thesis in sections 3.2.1 and 3.2.2, however, rely on hand-generated kernels provided by Tim Warburton. While hand-generating multiple highly performant kernels is an onerous task for the entire end-to-end solver, the two kernels considered in sections 3.2.1 and 3.2.2 are the most important kernels in the solver and therefore warrant the effort.

### 3.2.1 Matrix-Vector Product Kernel

The fast application of the matrix-vector product is a prerequisite for the construction of fast preconditioners for the spectral element Poisson problem. The matrix-vector product kernel, moreover, must be performance portable across several different GPU architectures, including the NVIDIA V100 and A100 GPUs as well as the AMD MI100 and MI250X GPUs. As several polynomial orders are required in applying the multigrid method described in section 2.6, the matrix-vector product kernel must be performant across all polynomial orders. Finally, the preconditioner is performed in single precision in order to increase the throughput of the solver. The matrix-vector product in the Krylov solver, however, is performed in double precision to ensure accuracy and convergence. Through Tim Warburton's efforts, several matrix-vector product kernels variants are developed. A brief description of each

kernel is given in table 3.3.

The baseline kernel described in table 3.3 already represents a well-optimized kernel for CUDA GPUs. The matrix-vector product kernels are taken from the `OCCA` implementation in `libParanumal` [69]. The kernel there includes the numerous optimizations described for the BK5 kernel in [70]. Included in these optimizations are the efficient use of shared memory with padding to avoid bank conflicts and careful use of register to reduce the number of global memory accesses while still reducing register pressure and preserving high occupancy.

Table 3.3: Summary of the strategies applied for $A\underline{x}$ kernel. $N_q = p + 1$ is the number of quadrature points in a single dimension. $n_D$ represents the number of dimensions in launching the thread block, while $t_0$, $t_1$, and $t_2$ represent the number of threads in each dimension. $E/b$ represents the number of elements in a thread block. For the cases where $E/b$ is not unity, a fixed number of elements are executed per each thread block. This specific number, however, is tuned for a given polynomial order, precision, and GPU.

| v | $E/b = 1$? | $n_D$ | $t_0$ | $t_1$ | $t_2$ | Description |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | $N_q$ | $N_q$ | 1 | Baseline kernel |
| 1 | 1 | 2 | $N_q$ | $N_q$ | 1 | Pad shmem to avoid bank conflicts, reduce register pressure |
| 2 | 0 | 3 | $N_q$ | $N_q$ | $E/b$ | Blocked element left-most index in shmem |
| 3 | 0 | 3 | $N_q$ | $N_q$ | $N_q \cdot E/b$ | 3D thread launch, only good for low orders |
| 4 | 0 | 3 | $N_q$ | $N_q$ | $E/b$ | Blocked element right-most index in shmem |
| 5 | 0 | 3 | $N_q$ | $N_q$ | $E/b$ | Same as 4, reduced register pressure |
| 6 | 1 | 2 | $N_q$ | $N_q$ | 1 | Same as 1, different shmem padding |

The performance of each kernel variant is measured, both on a single A100 GPU, as well as a single MI250X/1GCD GPU. The GFLOPS rate, fastest kernel, and speedup over the baseline kernel are reported in table 3.4 for the A100 GPU and table 3.5 for the MI250X/1GCD GPU. The results demonstrate that the fastest kernel variant depends on the polynomial order, precision, and even the GPU architecture. In addition, using the incorrect kernel variant can result in highly sub-optimal performance. Hard-coding the fastest kernel variant for each polynomial order, precision, and GPU architecture combination, however, is not a viable solution. This is especially the case for new and upcoming GPU architectures, such as the NVIDIA H100, AMD MI300, and Intel Ponte Vecchio GPUs. Therefore, run-time kernel selection is required for good performance portability. This is done by measuring the performance of each of the kernel variants described in table 3.3 and selecting the fastest kernel variant *at run-time*. This process is repeated for each polynomial order and precision required for the case. For example, a $(7, 3, 1)$ multigrid cycle requires the matrix-vector product kernel in single precision for polynomial order $p = 7$, $p = 3$, and $p = 1$, as well as double precision for $p = 7$.

To further illustrate the importance of run-time kernel selection, the performance of the

Table 3.4: $A\underline{x}$ kernel GFLOPS rates and speedup for A100

(a) Single precision

| $p$ | GFLOPS | Fastest Kernel | Speedup |
|---|---|---|---|
| 1 | 1428 | 2 | 3.43 |
| 2 | 1902 | 2 | 1.75 |
| 3 | 2493 | 2 | 1.18 |
| 4 | 2870 | 2 | 1.06 |
| 5 | 3323 | 4 | 1.27 |
| 6 | 3610 | 2 | 1.09 |
| 7 | 4244 | 4 | 1.02 |
| 8 | 4139 | 6 | 1.11 |
| 9 | 4556 | 4 | 1.14 |
| 10 | 4348 | 6 | 1.00 |
| 11 | 5015 | 6 | 1.07 |
| 12 | 4476 | 6 | 1.01 |

(b) Double precision

| $p$ | GFLOPS | Fastest Kernel | Speedup |
|---|---|---|---|
| 1 | 814 | 2 | 2.19 |
| 2 | 1009 | 4 | 1.25 |
| 3 | 1326 | 4 | 1.05 |
| 4 | 1504 | 2 | 1.02 |
| 5 | 1749 | 4 | 1.14 |
| 6 | 1913 | 2 | 1.07 |
| 7 | 2220 | 2 | 1.03 |
| 8 | 2220 | 2 | 1.10 |
| 9 | 2390 | 2 | 1.10 |
| 10 | 2460 | 2 | 1.09 |
| 11 | 2560 | 2 | 1.11 |
| 12 | 2341 | 2 | 1.17 |

Table 3.5: $A\underline{x}$ kernel GFLOP rates and speedup for MI250X/1GCD

(a) Single precision

| $p$ | GFLOPS | Fastest Kernel | Speedup |
|---|---|---|---|
| 1 | 1017 | 2 | 4.20 |
| 2 | 1101 | 5 | 1.58 |
| 3 | 1476 | 6 | 1.02 |
| 4 | 1668 | 1 | 1.73 |
| 5 | 2003 | 4 | 1.04 |
| 6 | 2522 | 6 | 2.49 |
| 7 | 3831 | 4 | 1.02 |
| 8 | 3246 | 5 | 3.40 |
| 9 | 3632 | 2 | 1.25 |
| 10 | 4000 | 4 | 4.20 |
| 11 | 4005 | 2 | 1.33 |
| 12 | 4165 | 5 | 4.61 |

(b) Double precision

| $p$ | GFLOPS | Fastest Kernel | Speedup |
|---|---|---|---|
| 1 | 491 | 2 | 2.76 |
| 2 | 504 | 5 | 1.08 |
| 3 | 762 | 2 | 1.17 |
| 4 | 949 | 5 | 1.07 |
| 5 | 1187 | 1 | 1.03 |
| 6 | 1317 | 4 | 1.00 |
| 7 | 1923 | 0 | 1.00 |
| 8 | 1742 | 4 | 1.05 |
| 9 | 1919 | 2 | 1.10 |
| 10 | 2167 | 0 | 1.00 |
| 11 | 2022 | 2 | 1.06 |
| 12 | 2233 | 2 | 1.36 |

matrix-vector product kernel is measured on a single V100 and MI250X/1GCD GPU in both single and double precision. The performance of the baseline kernel is compared against the performance of the fastest kernel variant for the given problem. The number of degrees of freedom is varied from a few hundred to 10 million, and the throughput performance is measured across a thousand trials. Double and single precision results for the V100 GPU are shown in figs. 3.1 and 3.2, respectively. The same for the MI250X/1GCD are shown in figs. 3.3 and 3.4. Work rates in terms of GFLOPS on the V100 GPU for double and single precision are shown in figs. 3.5 and 3.6; MI250X/1GCD GFLOPS results are in figs. 3.7

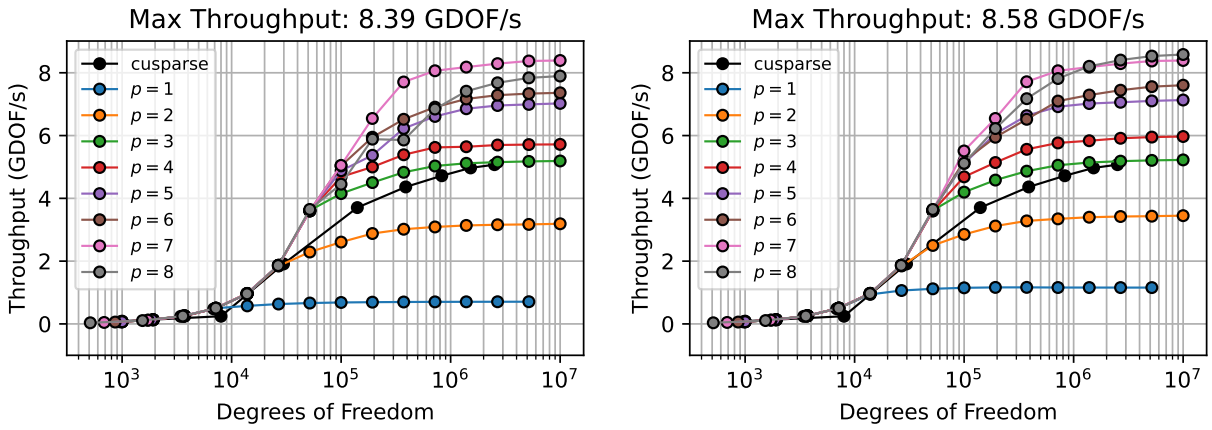



Figure 3.1: Single V100 throughput performance of $A\underline{x}$ kernel in 64-bit precision on Summit for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

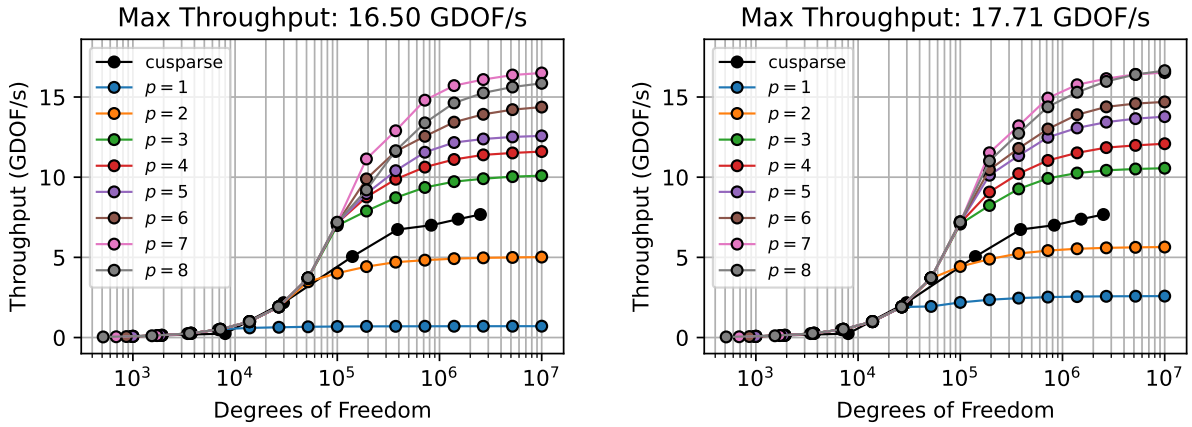



Figure 3.2: Single V100 throughput performance of $A\underline{x}$ kernel in 32-bit precision on Summit for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

and 3.8.

As shown in figs. 3.3 and 3.4, the maximum observed throughput is not significantly improved through run-time kernel selection. For the double precision $A\underline{x}$ kernel, the maximum observed throughput is 11.62 GDOF/s for the baseline kernel and 12.01 GDOF/s for the fastest kernel. Further, reasonably good performance is obtained across the considered polynomial orders, with exception to the $p = 1$ and $p = 2$ cases. For reference, the measured throughput using `hipsparse` on an assembled CSR matrix representing a three-dimensional finite-difference grid on a box is shown as the black line figs. 3.3 and 3.4. This observed

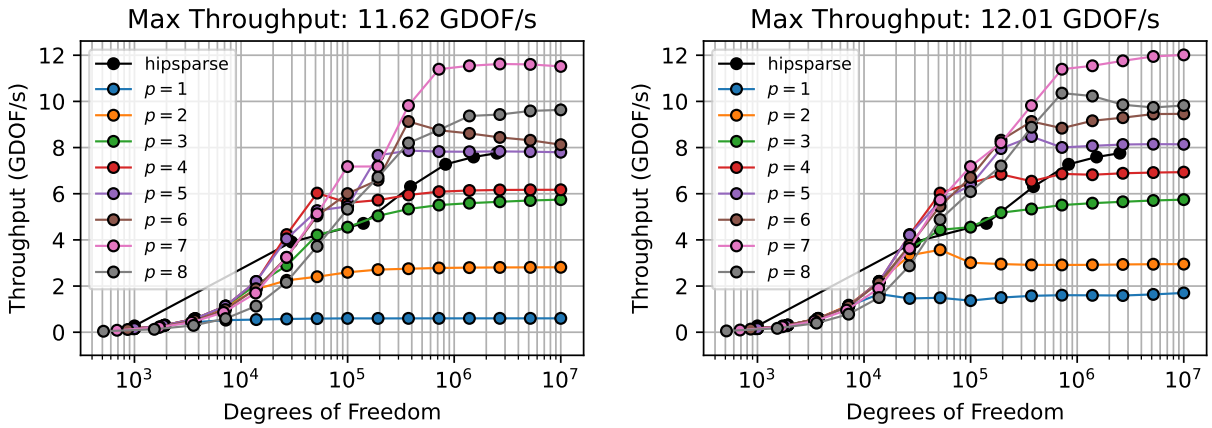



Figure 3.3: Single MI250X/1GCD throughput performance of $A\underline{x}$ kernel in 64-bit precision on Frontier for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

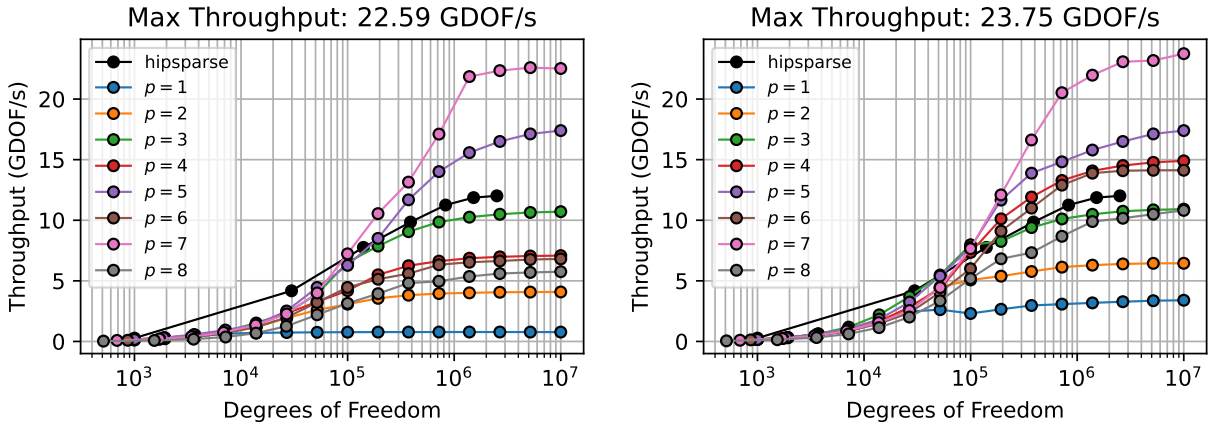



Figure 3.4: Single MI250X/1GCD throughput performance of $A\underline{x}$ kernel in 32-bit precision on Frontier for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

performance regression in the matrix-free implementation is explained at the end of this section.

Why does run-time kernel selection not improve the performance of the matrix-vector product kernel on the MI250X/1GCD for double precision? As noted previously, the baseline kernel is already highly optimized, especially for double precision. However, in single precision, the baseline kernel is not as well optimized for even values of $p$. The max throughput prior to tuning is 22.59 GDOF/s and 23.75 GDOF/s after tuning. Careful inspection, however, reveals that the throughputs prior to tuning are significantly lower for even values

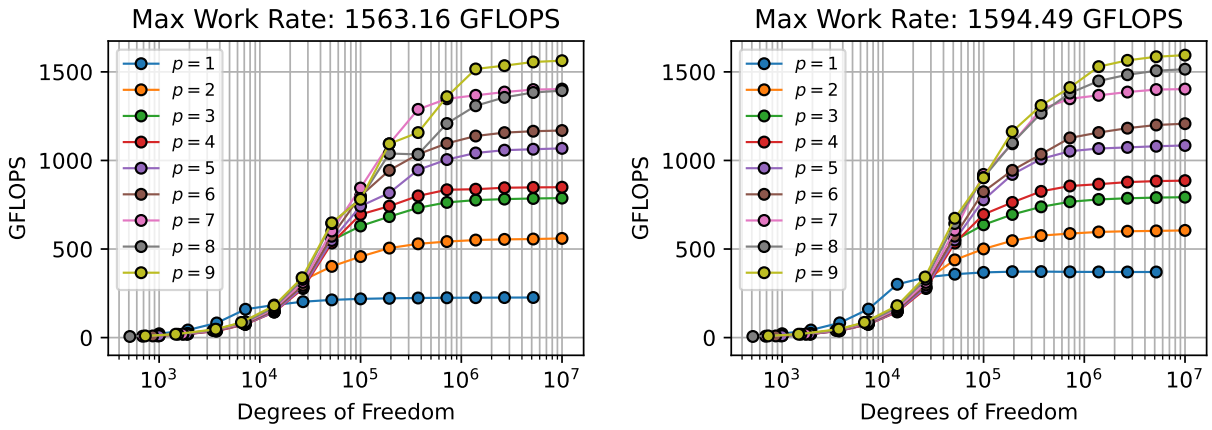



Figure 3.5: Single V100 GFLOPS performance of $A\underline{x}$ kernel in 64-bit precision on Summit for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

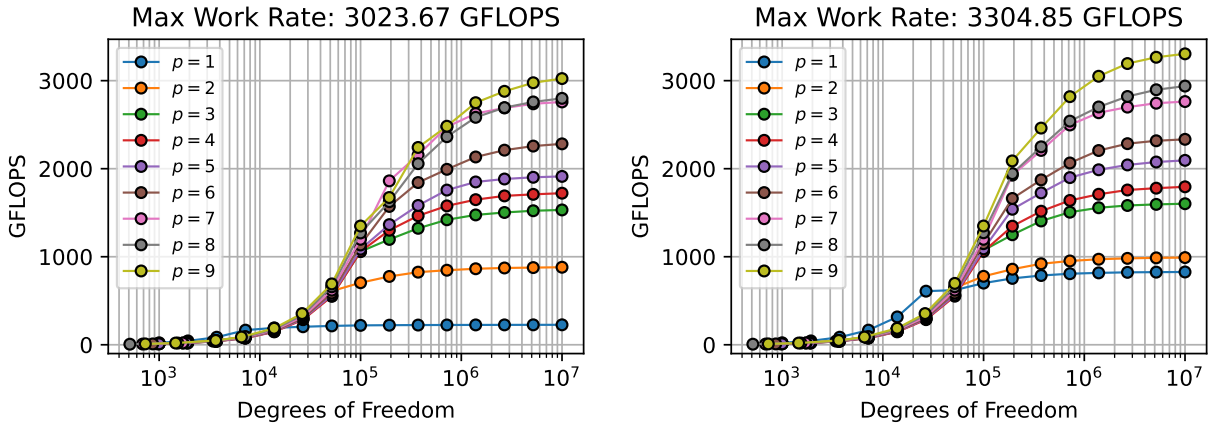



Figure 3.6: Single V100 GFLOPS performance of $A\underline{x}$ kernel in 32-bit precision on Summit for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

of $p$, such as $p = 4$ and $p = 6$. After tuning, however, these throughputs are significantly improved.

Similar results are observed for the V100 GPU, as shown in figs. 3.1 and 3.2. In double precision, the maximum throughput for the baseline kernel is 8.39 GDOF/s. while the run-time tuned kernel achieves 8.58 GDOF/s. In single precision, the maximum throughput for the baseline kernel is 16.50 GDOF/s and 17.71 GDOF/s for the run-time tuned kernel. An important observation is that, in single precision, the kernel throughputs are nearly a factor of two higher than in double precision. This is the case for both the MI250X/1GCD and

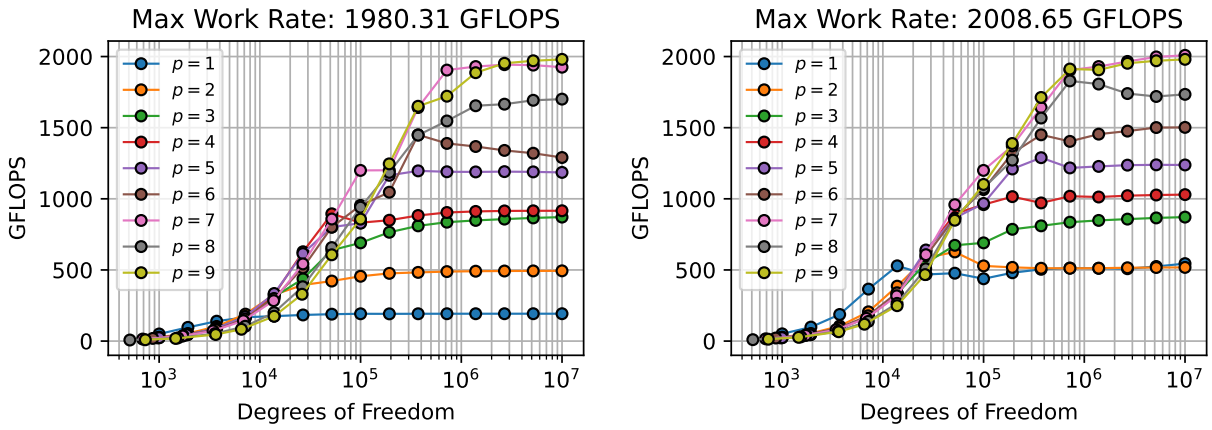



Figure 3.7: Single MI250X/1GCD GFLOPS performance of $A\underline{x}$ kernel in 64-bit precision on Frontier for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

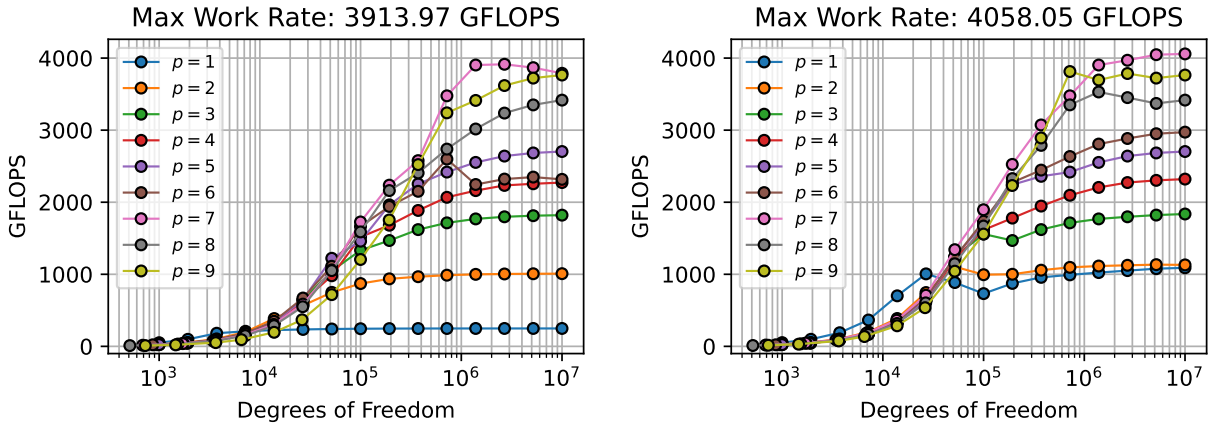



Figure 3.8: Single MI250X/1GCD GFLOPS performance of $A\underline{x}$ kernel in 32-bit precision on Frontier for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

V100 GPUs, motivating the use of mixed-precision for preconditioning.

In addition to the matrix-free $\underline{b} = A\underline{x}$ kernels, figs. 3.1 to 3.4. show the performance applying the assembled compressed sparse row (CSR) matrix through the native `cusparse` and `hipsparse` libraries. As noted by Jed Brown and coworkers in [71], it is possible to apply the matrix-free operator, even for $p = 1$, faster than the native CSR implementation. This, however, is not observed in these kernels as the solution vector is stored in E-vector format, see fig. 2.1, rather than the T-vector format used in the native CSR implementation. This results in roughly a factor of 8 increase in the words read from global memory for $\underline{x}$

and written to $\underline{b}$. However, unless smoothing at the $p = 1$ level is utilized, the matrix-free operator at $p = 1$ is never applied in the multigrid V-cycle preconditioner described in section 2.6. However, since the focus of this work is on the performance of the matrix-free operator for higher polynomial orders, we do not explore this further.

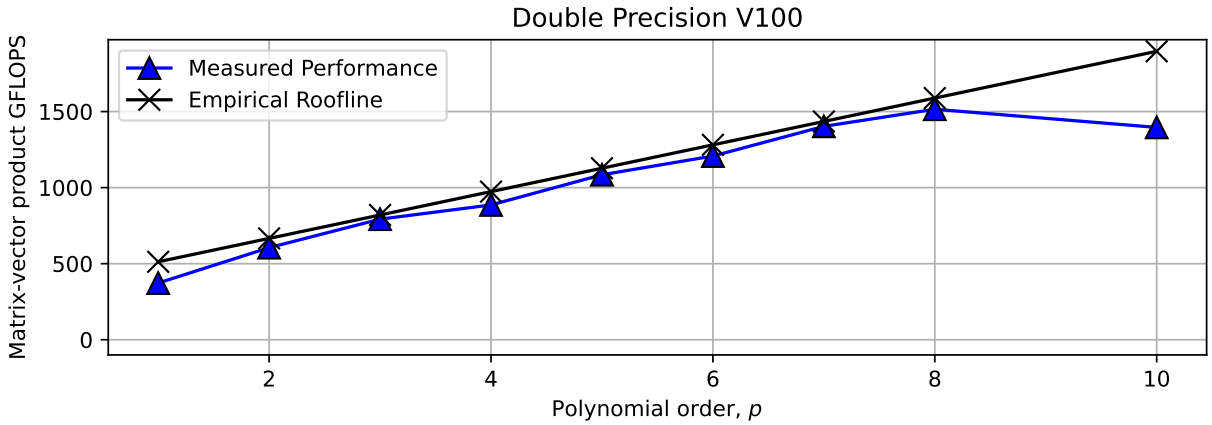



Figure 3.9: Single V100 measured maximum GFLOPS rate for the matrix-vector product kernel in 64-bit precision on Summit. The roofline model from eq. (3.6) is used with $B = 820$ GB/s.

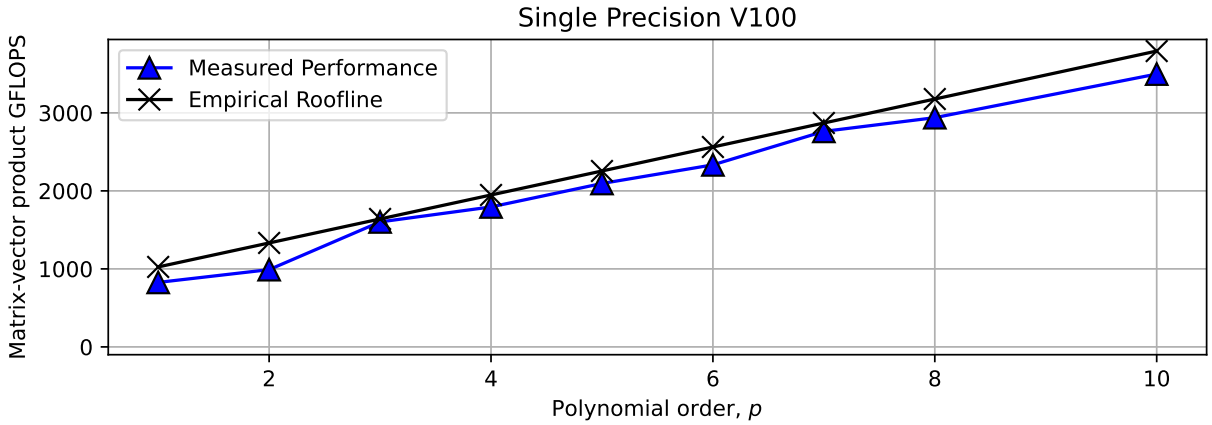



Figure 3.10: Single V100 measured maximum GFLOPS rate for the matrix-vector product kernel in 32-bit precision on Summit. The roofline model from eq. (3.6) is used with $B = 820$ GB/s.

While the kernel work rates and throughputs reported in this section are useful for understanding the performance of the matrix-free operator, is there an upper bound on the performance of the matrix-free operator? We perform a roofline model similar to [72]. The

computed rate, $R$, is modeled as

$$R = \min\left(C, \frac{B}{d_w + d_r} F\right), \tag{3.6}$$

where $C$ represents the peak work rate of the GPU in FLOPS, $B$ represents the peak (bidirectional) bandwidth in bytes per second, $d_w$ and $d_r$ represent the number of bytes written and read, respectively, and $F$ represents the number of FLOP invoked in the kernel. As $C$ is typically measured in several TFLOPS and $B$ is near 1 TB/s on current-generation GPU architectures, the relatively low arithmetic intensity of the matrix-free operator results in the bandwidth bound being the limiting factor. Based on the asymptotic streaming rates described in [73], we use here the same bandwidth values as in [72]: $B = 820$ GB/s for the V100 GPU and $B = 1117$ GB/s for the MI250X/1GCD GPU.

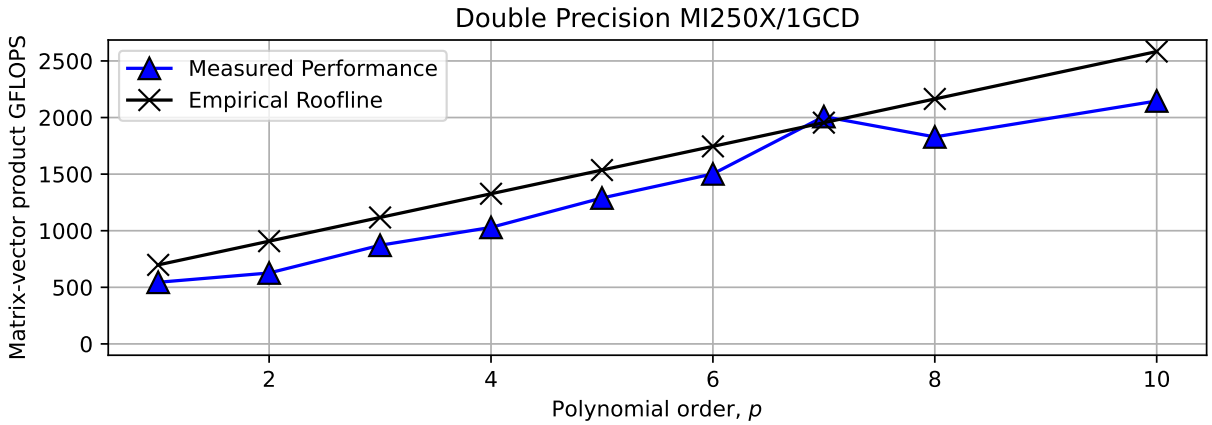



Figure 3.11: Single MI250X/1GCD measured maximum GFLOPS rate for the matrix-vector product kernel in 64-bit precision on Frontier. The roofline model from eq. (3.6) is used with $B = 1117$ GB/s.

With the roofline model described in eq. (3.6) available, we can compare the maximum observed work rate to the roofline model. This is done in figs. 3.9 and 3.10 for double and single precision on the V100 GPU, respectively, and in figs. 3.11 and 3.12 for double and single precision on the MI250X/1GCD GPU, respectively. As we observe, the maximum observed work rate matches the empirical roofline model across most polynomial orders. For particularly high polynomial orders, such as $N = 10$, we observe that the maximum observed work rate starts to deviate from the roofline model. Despite this, the maximum observed work rate remains within 80% of the empirical roofline model across all polynomial orders, precisions, and GPUs considered here.

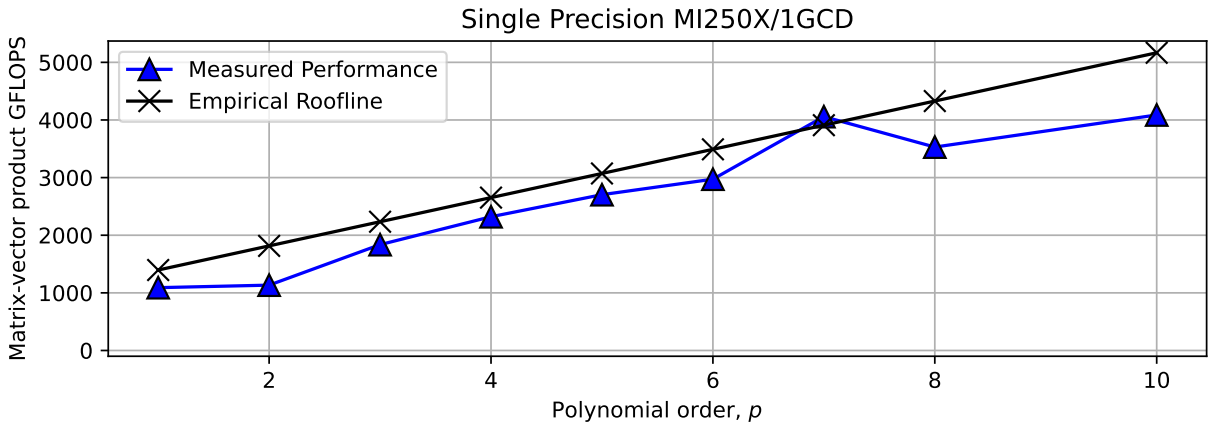



Figure 3.12: Single MI250X/1GCD measured maximum GFLOPS rate for the matrix-vector product kernel in 32-bit precision on Frontier. The roofline model from eq. (3.6) is used with $B = 1117$ GB/s.

### 3.2.2 Fast Diagonalization Method Kernel

The Schwarz smoothing strategies described in algs. 2.10 and 2.11 from section 2.6 require the application of the fast diagonalization method (FDM). Since this operation is performed as part of the smoother, potentially several times for the polynomial smoothers described in section 2.7, the performance of the FDM kernel is critical to the overall performance of the solver. As mentioned previously, this operation need only be applied in single precision, as it is merely a step in the preconditioner. Similar to the matrix vector product kernel in section 3.2.1, however, the FDM kernel must be performance portable across several different GPU architectures and polynomial orders. With help from Tim Warburton, several FDM kernel variants are developed. Table 3.6 describes each kernel variant.

Table 3.6: Summary of the strategies applied for FDM kernel. $N_q = p + 1$ is the number of quadrature points in a single dimension. $n_D$ represents the number of dimensions in launching the thread block, while $t_0$, $t_1$, and $t_2$ represent the number of threads in each dimension. $E/b$ represents the number of elements in a thread block. For the cases where $E/b$ is not unity, a fixed number of elements are executed per each thread block. This specific number, however, is tuned for a given polynomial order, precision, and GPU.

| v | $E/b = 1$? | $n_D$ | $t_0$ | $t_1$ | $t_2$ | Description |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | $N_q + 2$ | $N_q + 2$ | 1 | Baseline kernel |
| 1 | 0 | 3 | $N_q + 2$ | $N_q + 2$ | $E/b$ | Blocked element left-most index in shmem |
| 2 | 0 | 3 | $N_q + 2$ | $N_q + 2$ | $E/b$ | Blocked element right-most index in shmem |
| 3 | 0 | 3 | $N_q + 2$ | $N_q + 2$ | $E/b$ | Same as 2, reduce shmem usage |
| 4 | 1 | 2 | $N_q + 2$ | $N_q + 2$ | 1 | Same as 0, reduce shmem usage |

While baseline kernel described in table 3.6 offers decent performance, it is not optimal. The performance of each kernel variant is measured, both on a single V100 GPU, as well as a single MI250X/1GCD GPU in table 3.7. Note here that $p_e$ denotes the size of the overlapping subdomains, with $p_e = p+2$ representing the overlap scheme depicted in fig. 2.9. We observe that the fastest kernel variant provides a large speedup over the baseline kernel in all cases. Much like the matrix-vector product kernel in section 3.2.1, the fastest variant differs for each polynomial order and GPU architecture. Therefore, as before, the fastest kernel variant is selected at run-time to maximize the kernel throughput.

Table 3.7: FDM Kernel GFLOPS rates and speedup on A100 and MI250X/1GCD.

(a) Single precision A100 FDM kernel

| $p_e$ | GFLOPS | Fastest | Speedup |
|---|---|---|---|
| 3 | 2720 | 1 | 1.771 |
| 4 | 3787 | 4 | 1.607 |
| 5 | 4372 | 1 | 1.544 |
| 6 | 5211 | 4 | 1.662 |
| 7 | 5589 | 4 | 1.972 |
| 8 | 6938 | 4 | 1.721 |
| 9 | 7128 | 4 | 1.429 |
| 10 | 8261 | 4 | 1.748 |
| 11 | 8270 | 4 | 1.606 |
| 12 | 8272 | 4 | 1.785 |

(b) Single precision MI250X/1GCD FDM kernel

| $p_e$ | GFLOPS | Fastest | Speedup |
|---|---|---|---|
| 3 | 2746 | 2 | 2.657 |
| 4 | 3199 | 2 | 5.565 |
| 5 | 4153 | 3 | 1.751 |
| 6 | 4774 | 3 | 7.301 |
| 7 | 4347 | 4 | 1.525 |
| 8 | 5435 | 3 | 8.616 |
| 9 | 5019 | 3 | 1.84 |
| 10 | 5336 | 3 | 8.58 |
| 11 | 4736 | 3 | 1.993 |
| 12 | 5076 | 3 | 9.239 |

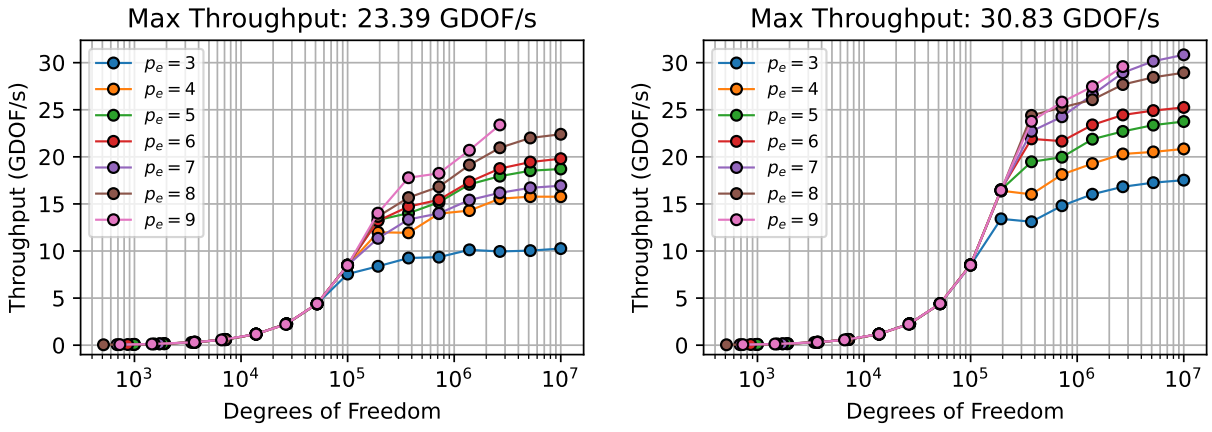



Figure 3.13: Single V100 throughput performance of FDM kernel in 32-bit precision on Summit for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

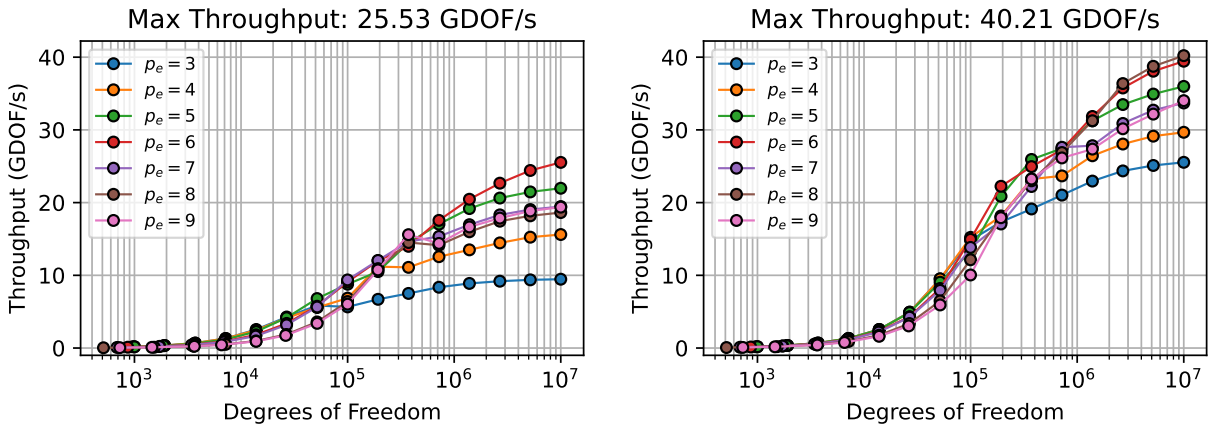



Figure 3.14: Single MI250X/1GCD throughput performance of FDM kernel in 32-bit precision on Frontier for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

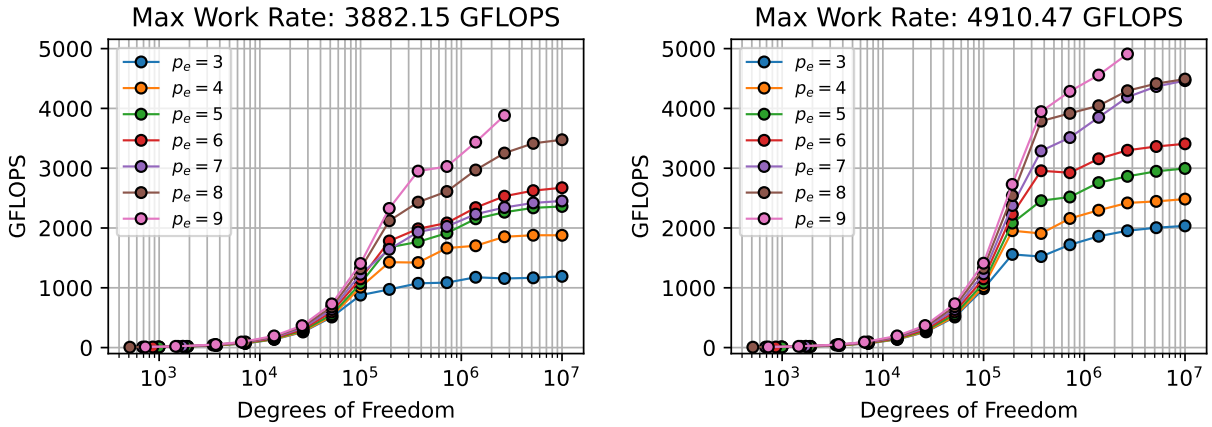



Figure 3.15: Single V100 GFLOPS performance of FDM kernel in 32-bit precision on Summit for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

Figure 3.13 shows the throughput performance of the FDM kernel before and after tuning on the V100 GPU. The same results in terms of GFLOPS are reported in fig. 3.15. The max throughput achieved prior to tuning is 23.39 GDOF/s. However, after tuning, the max throughput is 30.83 GDOF/s, a 31.8% increase in performance. A much larger performance boost is observed on the MI250X/1GCD GPU. As shown in fig. 3.14, the max throughput is raised from 25.53 GDOF/s to 40.21 GDOF/s, a 57.6% increase in performance. An additional benefit from the run-time kernel selection is that the measured throughputs are more consistent across polynomial orders. For example, an odd-even pattern is observed in

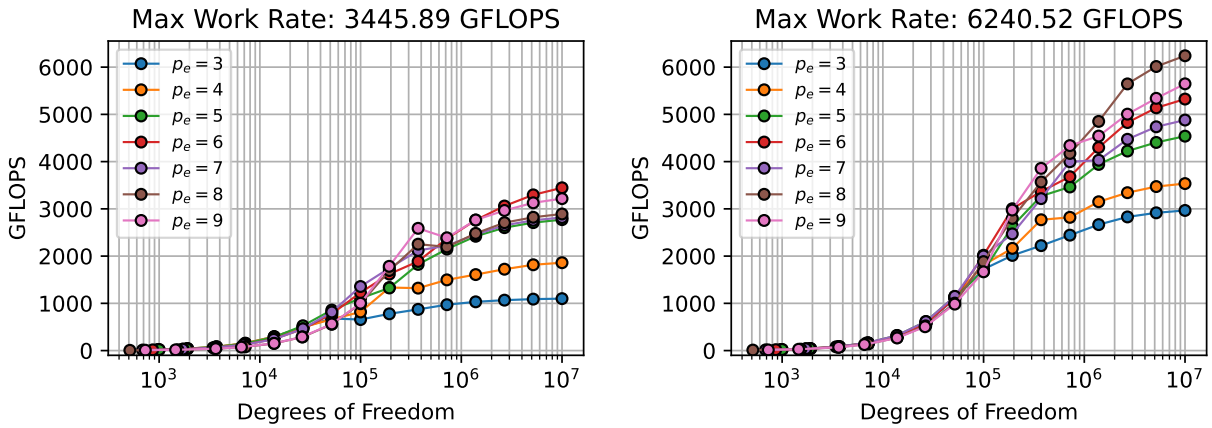



Figure 3.16: Single MI250X/1GCD GFLOPS performance of FDM kernel in 32-bit precision on Frontier for varying polynomial orders and degrees of freedom. Left: no run-time kernel selection, right: run-time kernel selection.

the left in fig. 3.14, where the throughput is much lower for even polynomials orders. This is mitigated through the run-time kernel selection, however. For this reason, run-time kernel selection, both for the FDM and matrix-vector product kernel in section 3.2.1, is used for the remainder of this work.

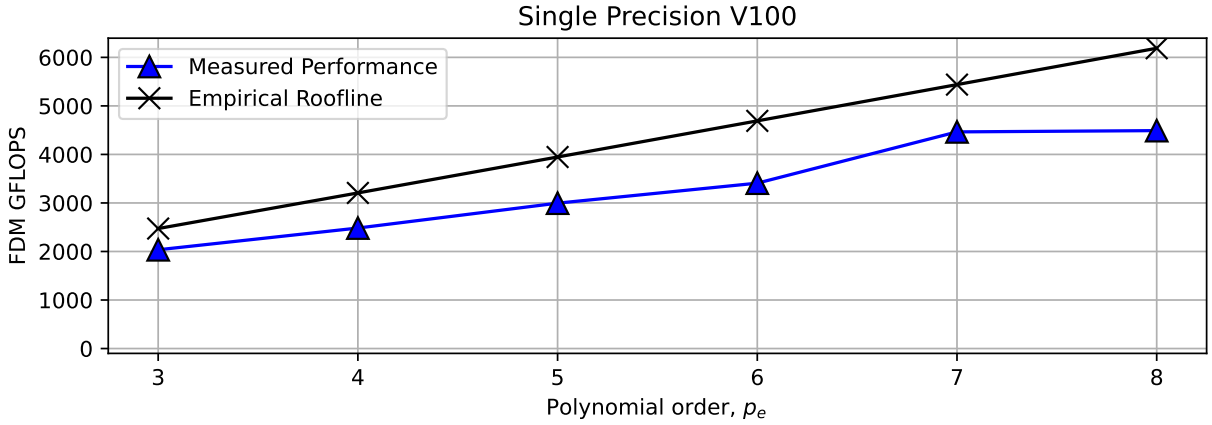



Figure 3.17: Single V100 measured maximum GFLOPS rate for the FDM kernel in 32-bit precision on Summit. The roofline model from eq. (3.6) is used with $B = 820$ GB/s.

Is it possible to further improve the performance of the FDM kernel? According to the roofline model from eq. (3.6), the answer is not by much. Figure 3.17 shows the measured maximum GFLOPS rate as compared to the empirical roofline model for the V100 GPU with $B = 820$ GB/s. The same results for the MI250X/1GCD with $B = 1117$ GB/s are shown in fig. 3.18. The difference between the observed maximum work rate and the roofline

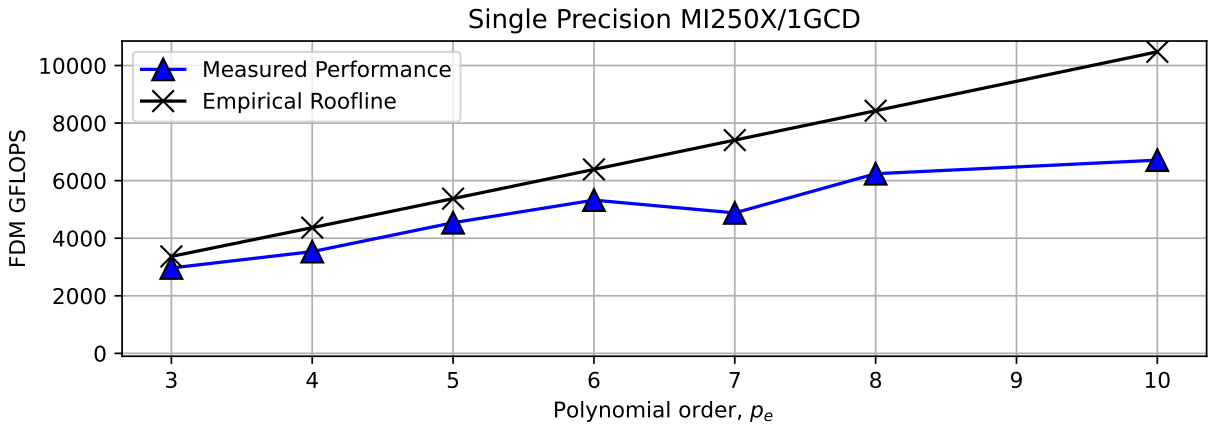



Figure 3.18: Single MI250X/1GCD measured maximum GFLOPS rate for the FDM kernel in 32-bit precision on Frontier. The roofline model from eq. (3.6) is used with $B = 1117$ GB/s.

model is larger for the FDM kernel than the matrix-vector product kernel in section 3.2.1. However, the FDM kernel still achieves roughly 80% of the roofline model across all but the highest polynomial orders on both the V100 and MI250X/1GCD GPU. We observe that the matrix-vector product kernel consistently achieves at least 80% to 90% of the roofline model across all polynomial orders, precisions, and GPUs. We further observe that the FDM kernel generally reaches 80% of the roofline model for most polynomial orders, irrespective of the GPU. As such, we conclude that both kernels are nearly optimal. We therefore now focus our attention on the performance of the various solver components *in parallel* in section 3.3.

## 3.3 UNDERSTANDING THE COST OF VARIOUS OPERATORS

Important preconditioner components include the matrix-vector product, the additive Schwarz smoother in alg. 2.10 from section 2.6, a boomerAMG V-cycle on the GPU to approximate the low-order operator, $A_F^{-1}$ from section 2.4, and a coarse-grid solve $A_c^{-1}$ on the CPU as required in algs. 2.8 and 2.9. These relatively simple operators form the basis of the preconditioners considered in this thesis; all preconditioners considered in the current work are the composition of these operators, albeit at different polynomial orders. Therefore, it is imperative that these operators be as fast possible and performance-portable. Run-time kernel selection, as described in section 3.2.1 and section 3.2.2 for the matrix-vector product and FDM kernels, respectively, is one technique to ensure performance portability.

To illustrate the relative cost of these various components of the Poisson solver, a weak scaling study for the Kershaw ($\varepsilon = 0.05$) benchmark problem (see section 2.8.1) is performed

with $n/P \sim 2.67M$ unknowns per V100 GPU, starting at $P = 6$ GPUs. The time to apply $A$, $S_{ASM}$, and a single boomerAMG V-cycle to approximate $A_F^{-1}$ on the GPU and $A_c^{-1}$ on the CPU is shown in fig. 3.19. For more detail regarding $S_{ASM}$, see section 2.6. In addition, fig. 3.19 also shows the time to compute 10 weighted inner products (wdotp), including the all-reduce across all processors. Despite the higher kernel throughputs observed in the

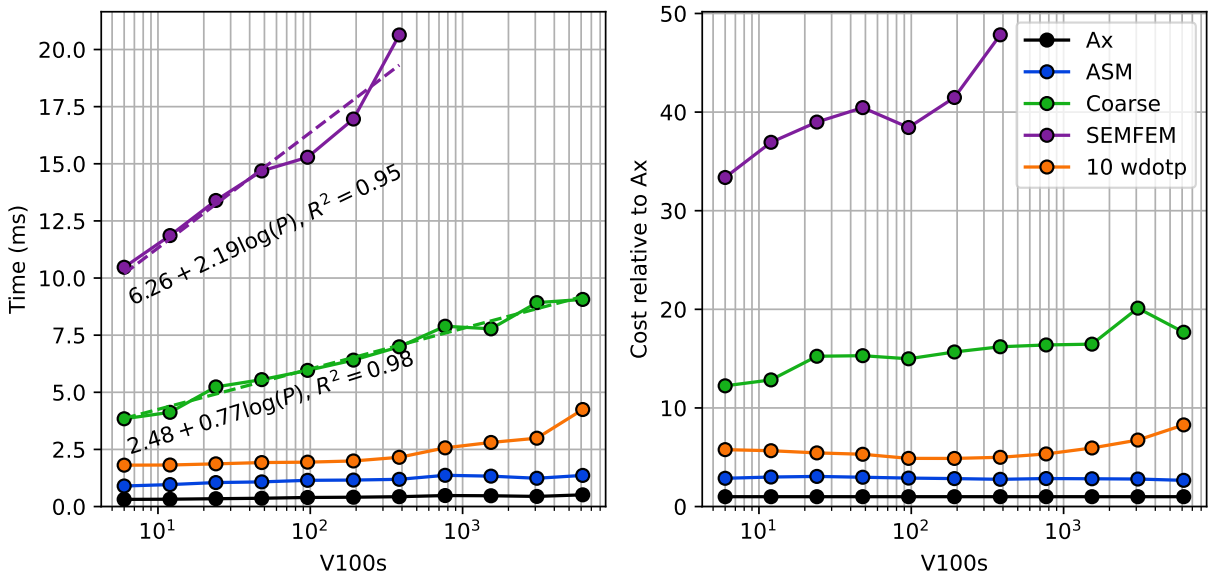



Figure 3.19: Weak scaling operator cost study for the Poisson solver for the Kershaw benchmark problem, $\varepsilon = 0.05$.

FDM kernel application (fig. 3.13) compared to the matrix-vector product kernel (fig. 3.2), the total time to apply $S_{ASM}$ is nearly a factor three more expensive due to the increased number of *gather-scatter* applications required, see alg. 2.10. Both of these operations, which require only highly-scalable *gather-scatter* communication operations, exhibit practically no $P$-dependence in the time to apply them. This is the case even when moving from entirely *on-node* communication at $P = 6$ to *off-node* communication. The relatively expensive coarse-grid solve and SEMFEM operator costs, however, scales as $\log P$. For SEMFEM, this observation is consistent with the performance cost analysis performed by Fischer and coworkers [74] for geometric multigrid. Coarse-grid solves, moreover, scale as $\log P$ due to the all-to-all communication pattern required in the solution of the coarse-grid problem [75, section 1.4]. As we will discuss in section 3.3.1, the communication costs associated with the coarse-grid solve are especially expensive in comparison to the relatively fast compute provided on heterogeneous architectures.

From fig. 3.19, why is applying the low-order AMG V-cycle much more expensive compared

to the matrix-vector product and smoother applications, even at relatively *low* processor counts. Let us consider applying the AMG V-cycle for approximating $A_F^{-1}$. The AMG grid complexity, which measures the number of non-zeros per row in the entire multigrid V-cycle relative to the finest grid, remains approximately 2.56 for the problem considered in fig. 3.19. As a very light-weight $L_1$ Jacobi smoother is employed, the equivalent work of two matrix-vector products per cycle per level are performed. Assuming the throughput of applying the low-order operator is comparable to applying the high-order operator, this equates to the expense of roughly 5.12 matrix-vector products per V-cycle. However, even at $P = 6$ processors, the measured time to apply the AMG V-cycle is approximately 30 times larger than a matrix-vector product. There are two primary reasons for this discrepancy. First, the assembled CSR matrix-vector product throughput steeply drops off with respect to the number of degrees-of-freedom in the operator, as observed in the assembled matrix-vector product throughputs from section 3.2.1. Second, each algebraic multigrid level implies a necessary communication in applying the operator. This grows from 8 multigrid levels for $P = 6$ to 11 levels for $P = 768$.

### 3.3.1 Communication Cost on Heterogeneous Architectures

The following communication analysis is from [74, 75]. To understand the effect of communication on the performance of our solvers, we consider the communication cost in terms of the available work while a single word is communicated. Given the inter-node latency in seconds, $\alpha^*$; the inverse-bandwidth as seconds per word, $\beta^*$; the total number of words to be communicated, $m$; and the time in seconds to send the message, $t_c$; the communication cost is given by the following model:

$$t_c(m) = \alpha^* + \beta^* m. \tag{3.7}$$

Introducing an inverse throughput measure in terms of seconds per FLOP, $t_a$, the following non-dimensionalization of eq. (3.7) is obtained:

$$t_c(m) = (\alpha + \beta m) t_a. \tag{3.8}$$

Table 3.8 shows empirically measured machine-performance characteristics for various machines. Reproduced from [75], table 3.8 is constructed using the table from [74] with Summit values measured from [76] and the Summit throughput from [77].

From table 3.8, we observe that the heterogeneous Summit architecture is significantly

Table 3.8: Empirical machine-dependent parameters over a range of supercomputers. For reference, OLCF's Summit are equipped with 6 V100 GPUs per node, with a theoretical peak performance of 7 TFLOPS per GPU. In CFD simulations, however, a more realistic work rate of 0.5 TFLOPS per GPU is utilized. $m_2 := \alpha^*\beta^*$ is the message size at which it would take twice as long to send as a single word message.

| Year | $t_a$ ($\mu$s) | $\alpha t_a$ ($\mu$s) | $\beta t_a$ ($\mu$s/wd) | $\alpha$ | $\beta$ | $m_2$ | machine |
|---|---|---|---|---|---|---|---|
| 1986 | 50 | 5960 | 64 | 119.2 | 1.28 | 93 | Intel iPSC-1 (286) |
| 1987 | 0.333 | 5960 | 64 | 17898 | 192 | 93 | Intel iPSC-1/VX |
| 1988 | 10 | 938 | 2.8 | 93.8 | 0.28 | 335 | Intel iPSC-2 (386) |
| 1989 | 0.25 | 938 | 2.8 | 3752 | 11.2 | 335 | Intel iPSC-2/VX |
| 1990 | 0.1 | 80 | 2.8 | 800 | 28 | 29 | Intel iPSC-i860 |
| 1991 | 0.1 | 60 | 0.8 | 600 | 8 | 75 | Intel Delta |
| 1992 | 0.066 | 50 | 0.15 | 760 | 2.3 | 333 | Intel Paragon |
| 1995 | 0.02 | 60 | 0.27 | 3000 | 13.5 | 222 | IBM SP2 (BU96) |
| 1996 | 0.016 | 30 | 0.02 | 1875 | 1.25 | 1500 | ASCI Red 333 |
| 1998 | 0.006 | 14 | 0.06 | 2333 | 10 | 233 | SGI Origin 2000 |
| 1999 | 0.005 | 20 | 0.04 | 4000 | 8 | 500 | Cray T3E/450 |
| 2005 | 0.002 | 4 | 0.026 | 2000 | 13 | 154 | BGL/ANL |
| 2008 | 0.0017 | 3.5 | 0.022 | 2060 | 13 | 160 | BGP/ANL |
| 2011 | 0.0007 | 2.5 | 0.002 | 3570 | 2.87 | 1250 | Cray Xe6 (KTH) |
| 2012 | 0.0007 | 3.8 | 0.0045 | 5430 | 6.43 | 845 | BGQ/ANL |
| 2015 | 0.0004 | 2.2 | 0.0015 | 5500 | 3.75 | 1467 | Cray XK7 |
| 2020 | 0.000002 | 12.6 | 0.0011 | 6300000 | 550 | 11456 | Summit |

different from the other machines considered, in that millions of operations can be performed within the span of a single communication. This, in turn, implies that the per GPU workload must be sufficiently large to enable good performance. This is true both from the fact that sufficient work must be present to reach good kernel throughputs, as shown in section 3.2, as well as the fact that the communication cost relative to FLOP cost is much larger on heterogeneous architectures than traditional CPU-only machines. This is the reason we observe the strong scaling characteristics described in section 2.9. For example, the strong scaling limits on Summit are established at no less than $n/P \sim 2M$ gridpoints per GPU.

How can we utilize the fact that the problem sizes per GPU must be relatively large to our advantage? For the matrix-vector product and Schwarz operations shown in fig. 3.19, only sparse point-to-point communication is required. Given that this communication scales with the surface area of the domain owned by a given processor, the square-cube law relationship between the surface area and volume of the per-processor domain is exploited. The ratio of the per-processor volume, which dictates the total amount of work required, to the per-processor surface area *must* be larger on heterogeneous architectures than traditional

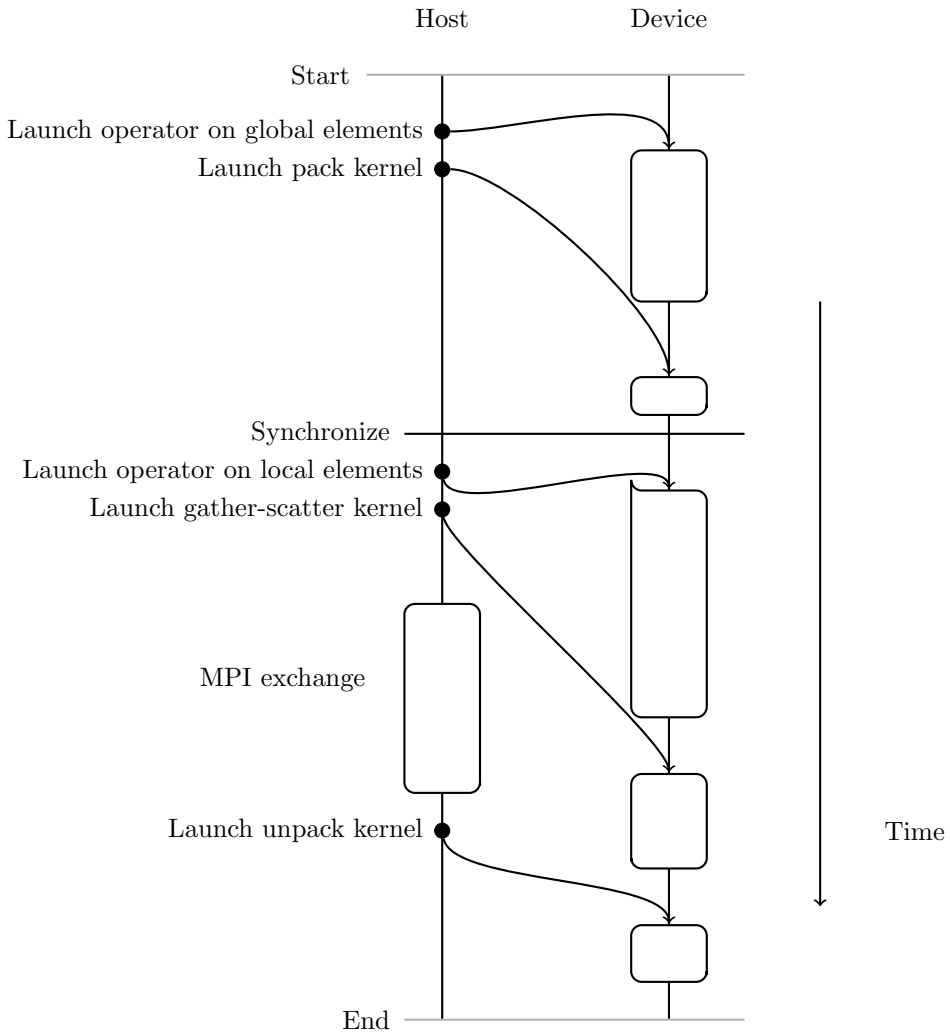



Figure 3.20: Timeline visualization for overlapping communcation and computation. This allows `nekRS` to effectively hide the communication cost associated with the matrix-vector product and Schwarz smoother.

CPU-architectures. Since there are so many more elements that lie entirely within the per-processor volume, we can effectively overlap much of the communication and computation required in the matrix-vector product and Schwarz operations.

In `nekRS`, a communication/computation overlapping strategy similar to those considered by Chalmers and coworkers in [72] is employed. Figure 3.20 shows a timeline visualization of the overlapping strategy employed in `nekRS`. First, the surface-elements that lie at inter-processor boundaries are executed in the operator kernel evaluation. Once complete, those entries are packed into send buffers. Afterward, the operator kernel is launched over the interior "local" elements, allowing a large portion of the computation to be overlapped with

the communication. The MPI exchange in question is implemented using either `MPI_Isend` and `MPI_Irecv` following by a `MPI_Waitall` call (referred to as "pair-wise" exchange), or through a sparse `MPI_Neighbor_alltoallv` (referred to as "neighbor all-to-all" exchange).

*A priori*, the optimal communication pattern is not known until run-time. To further complicate this matter, GPU-aware MPI implementations may execute slower than transferring the data from the GPU to the host, performing the communication on the host, and then finally transferring the data back to the device. This may occur especially when the *volume* of words communicated is small—in that regime, the expense associated with offloading the data back to the host may be comparatively small. To address this, the handle associated with each *gather-scatter* operation in `nekRS` is tuned at run-time. The fastest combination of pair-wise or neighbor all-to-all, performing the pack/unpack operations on the CPU or GPU, and whether to perform the MPI communication with the GPU buffer or transfer to the host is determined. This ensures that the handle associated with each *gather-scatter* occurs as quickly as possible, as it is the fundamental communication pattern required in the matrix-vector product and Schwarz smoother operations, which are needed at several different polynomial orders when used as part of the $p$-multigrid preconditioner.

The relatively poor parallel scalability of the coarse-grid and AMG V-cycle operators as compared to the matrix-vector product and Schwarz operators from fig. 3.19 is due to the increased number of communication operations required in the former two. Both the coarse-grid operator and AMG V-cycle associated with the low-order preconditioner require a global reduction that scales as $\log P$. To further worsen matters, at each algebraic multigrid level, the assembled CSR matrix-vector product requires an additional nearest-neighbor exchange. There are, however, several strategies that can be employed to mitigate this cost. First, through improving the low-order operator by treating it as a coarse-sparse in an auxiliary space method, the number of iterations required to converge can be reduced. This implies fewer AMG V-cycle calls needed to reach convergence. This idea is covered in chapter 6. A technique to mitigate the relatively poor parallel scalability of the coarse-grid solver is through overlapping it with the remainder of the V-cycle. This topic is the subject of chapter 5. Finally, another technique to ensure that fewer coarse-grid solves are required is to invest in better multigrid smoothers. Chapter 4 discusses polynomial smoothers which are asymptotically optimal as the polynomial smoother order is increased.

## Chapter 4: Optimal Smoothing Polynomials and One-Sided V-Cycles

In section 2.7, polynomial smoothers based on the shifted and scaled first-kind Chebyshev polynomials are constructed as the solution to a mini-max problem on the interval $[\lambda_{min}, \lambda_{max}]$. While the Chebyshev polynomials of the first-kind provide good smoothing properties when paired with the Schwarz-based (section 2.6) or point-wise Jacobi smoothing, the *ad-hoc* $\lambda_{min}$ parameter must be chosen. While the sensitivity analysis from table 2.2 indicates that good solver performance can be achieved for a wide-range of $\lambda_{min}$ values, tuning the $\lambda_{min}$ parameter can yield a 10-20% reduction in the iteration count. This analysis, further, is only conducted for a single problem with $k = 3$ order smoothing. In this chapter, we will show, utilizing results from Lottes [23], that smoother polynomials can be constructed as the solution to theoretical multigrid error bounds. Notably, theoretical results based on the value of a multigrid error bound demonstrate that the maximum eigenvalue of the multigrid error propagator scales as $O(k^{-1})$ as the order $k \to \infty$ when using the first-kind Chebyshev smoothers from section 2.7 with a fixed value of $\lambda_{min}$. However, we achieve $O(k^{-2})$ scaling in the maximum eigenvalue of the multigrid error propagator through the polynomial smoothers considered in this chapter.

The organization of this chapter is as follows. Section 4.1 introduces polynomial smoothers based on Chebyshev polynomials of the fourth kind which are developed by Lottes in [23]. Lottes's contributions from [23] are summarized in section 4.1.1. The remaining content in this chapter, however, are original contributions in [78] and this thesis. While the first-kind Chebyshev smoothers solve a mini-max problem on the interval $[\lambda_{min}, \lambda_{max}]$, the fourth-kind Chebyshev smoothers solve a *weighted* mini-max problem posed from a two-level error bound posed by Hackbusch [79]. A multi-level V-cycle error bound due to Lottes [23] is used to construct polynomial smoothers that are optimal with respect to that error bound. A related question we answer in section 4.1 is the optimal way of distributing a fixed number of smoothing operations $m + n = 2k$, with $m$ pre-smoothing and $n$ post-smoothing operations. In section 4.2, the various polynomial smoothers are compared for the Poisson equation discretized with 2D finite differences, and shown to have good agreement with the theoretical results from section 4.1. The quality of the error-bound from [23] is also investigated, and found to not be as sharp as desired. A local Fourier analysis (LFA) method extending the work of Thompson and coworkers in [47] to the new polynomial smoothers is presented in section 4.3. While this provides sharper estimates for the error bound, it is still not as sharp as desired. Numerical results in the context of matrix-free $p$-multigrid methods are considered in section 4.4. Finally, to address the lack of sharpness in the error bound,

smoother polynomials are constructed directly to optimize the convergence rate solver in section 4.5. For this, the fourth-kind Chebyshev polynomials are used as a basis to construct polynomial smoothers optimized through Nelder-Mead optimization.

## 4.1 OPTIMAL POLYNOMIAL SMOOTHERS FOR MULTIGRID

In this section, we describe the work by Lottes [23] on optimal polynomial smoothers for multigrid in section 4.1.1. With the fourth and optimized fourth-kind Chebyshev smoothers described, we then extend Lottes's analysis to the first-kind Chebyshev polynomial smoothers in section 4.1.2. In addition, section 4.1.2 applies a V-cycle error bound developed by Lottes to determine the optimal way of distributing smoother passes between the pre- and post-smoothing phases of the multigrid V-cycle.

### 4.1.1 Fourth and Optimized Fourth-Kind Chebyshev Polynomials

Is it possible to further improve the polynomial smoother and remove the ad-hoc $\lambda_{min}$ parameter from the 1st-kind Chebyshev smoother, as described in section 2.7? Following the work by Lottes [23], it is possible to improve the quality of the polynomial smoothers. The polynomial smoother can be chosen in such a way to minimize an error bound [23], such as the *two-level* bound proposed by Hackbusch [79] in eq. (4.1). Without loss of generality, let $\rho(SA) = 1$. Let $G_k(SA)$ be a $k$-order polynomial in $SA$. The *two-level* bound from Hackbusch [79] is re-written as [23]:

$$\begin{aligned} \|E_{\searrow}\|_A &= \left\|(I - PA_c^{-1}P^TA)G_k\right\|_A \\ &\leq C_0^{1/2} \sup_{0<\lambda\leq 1} \lambda^{1/2}|p_k(\lambda)|, \end{aligned} \tag{4.1}$$

where $C_0$ is the multigrid approximation property constant, which for a given level $j$ is

$$\begin{aligned} C_j &:= \left\|A_j^{-1} - P_{j+1}^j A_{j+1}^{-1}\left(P_{j+1}^j\right)^T\right\|_{A_j,S_j}^2 \\ &:= \sup_{\|\underline{f}\|_{S_j}\leq 1} \left\|\left(A_j^{-1} - P_{j+1}^j A_{j+1}^{-1}\left(P_{j+1}^j\right)^T\right)\underline{f}\right\|_{A_j}^2. \end{aligned} \tag{4.2}$$

Solving the *weighted* mini-max problem in eq. (4.1) yields the solution [23]:

$$p_k(\lambda) = \frac{1}{2k+1} W_k(1-2\lambda), \tag{4.3}$$

where $W_k$ is the Chebyshev polynomial of the 4th-kind of order $k$. Chebyshev polynomials of the 4th-kind satisfy the same recurrence as that of the first-kind, with different initial conditions [23, 80]:

$$W_n(x) = 2xW_{n-1}(x) - W_{n-2}(x) \text{ with } W_0(x) = 1, W_1(x) = 2x + 1. \tag{4.4}$$

While the fourth-kind Chebyshev polynomials optimize the two-level bound from Hackbusch [79] in eq. (4.1), can we construct a polynomial that is optimal with respect to a V-cycle bound? This is precisely what Lottes considers in [23] through optimizing the error-bound presented in lemma 4.1.

**Lemma 4.1.** Let the smoother iteration (on each level $j$) be given by

$$G_j = p_{k_j}(S_j A_j) \tag{4.5}$$

where $S_j$ is SPD and scaled such that $\rho(S_j A_j) = 1$, and $p_{k_j}(x)$ is a $k_j$-order polynomial satisfying $p_{k_j}(0) = 1$ and $|p_{k_j}(x)| < 1$ for $0 < x \leq 1$, possibly different on each level. Then the V-cycle contraction factor

$$\|E_{\searrow}\|_A^2 \leq \max_{j \in 0,\ldots,\ell-1} \frac{C_j}{C_j + \gamma_j^{-1}} \tag{4.6}$$

where $C_j$ is the approximation property constant for level $j$, defined in eq. (4.2), and

$$\gamma_j = \sup_{0<\lambda\leq 1} \frac{\lambda\, p_{k_j}(\lambda)^2}{1 - p_{k_j}(\lambda)^2}. \tag{4.7}$$

In order to optimize the bound in lemma 4.1, Lottes considers the fourth-kind Chebyshev polynomials as a basis for the polynomial smoother given by

$$p_k(\lambda) = \sum_{i=0}^{k} \frac{\beta_i - \beta_{i+1}}{2i+1} W_i(1 - 2\lambda), \tag{4.8}$$

with $\beta_0 = 1$ and $\beta_{k+1} = 0$. For the standard 4th-kind Chebyshev polynomial, $\beta_i := 1$ for all $i$, except for $\beta_0 = 1$ and $\beta_{k+1} = 0$. In the case of the optimized fourth-kind Chebyshev smoother, however, the $\beta_i$ coefficients can be chosen to form a polynomial smoother that is optimal in the error bound from lemma 4.1 [23].

Given the polynomial smoother described in eq. (4.8) and relaxing the $\rho(SA) = 1$ assumption, a similar iterative algorithm to alg. 2.12 can be derived as alg. 4.1. The inclusion

of the $\beta_i$ parameters in alg. 4.1 comes as a result of optimizing eq. (4.7) in the context of the *multi-level* error bound from lemma 4.1 [23] [4] . For the standard 4th-kind Chebyshev polynomial, $beta_i = 1$ for all $i$ in alg. 4.1. While not a Chebyshev polynomial, the polynomial optimizing the error bound in lemma 4.1 has a convenient basis in the 4th-kind Chebyshev polynomials. The simple smoothing iteration with $\omega = 3/2$, 1st-kind Chebyshev smoothing with $\lambda_{min} = 0.3\lambda_{max}$, and the 4th-kind Chebyshev smoothing polynomials, with and without optimal $\beta_i$, are shown in fig. 4.1. Minimizing eq. (4.7) with respect to $\lambda_{min}$ in the 1st-kind Chebyshev smoothing iteration, a contribution considered in this thesis, is also shown.

**Algorithm 4.1:** Chebyshev smoother, (Opt.) 4th-kind

**Data:** Initial solution, $\underline{x}_0$; Chebyshev polynomial order, $k$
**Result:** Smooth solution, $\underline{x}_k$
$\underline{x}_0 = \underline{x}$
$\underline{r}_0 = \underline{b} - A\underline{x}_0$ `/* Omit` $A\underline{x}_0$ `if` $\underline{x}_0 = \underline{0}$ `*/`
$\underline{d}_0 = \frac{4}{3}\frac{1}{\lambda_{max}} S\underline{r}_0$
**for** $i = 1, \ldots, k-1$ **do**
  $\underline{x}_i = \underline{x}_{i-1} + \beta_i \underline{d}_{i-1}, \underline{r}_i = \underline{r}_{i-1} - A\underline{d}_{i-1}$
  $\underline{d}_i = \frac{2i-1}{2i+3}\underline{d}_{i-1} + \frac{8i+4}{2i+3}\frac{1}{\lambda_{max}} S\underline{r}_i$
**end**
$\underline{x}_k = \underline{x}_{k-1} + \beta_k \underline{d}_{k-1}$

How does the $\gamma^{-1}$ term from the bound in lemma 4.1 scale with respect to $k$ for the various polynomial smoothers? Lottes [23] notes that, for the simple smoother iteration with weight $\omega$,

$$\gamma_s^{-1}(k) = 2\omega k. \tag{4.9}$$

The 4th-kind Chebyshev polynomial, however, has

$$\gamma_4^{-1}(k) = \frac{4}{3}k(k+1), \tag{4.10}$$

while the optimized 4th-kind Chebyshev polynomial is

$$\gamma_{4_{opt}}^{-1}(k) = \frac{4}{\pi^2}(2k+1)^2 - \frac{2}{3}. \tag{4.11}$$

Most notably, both 4th-kind Chebyshev polynomials types improve the $O(k)$ simple smoother iteration to $O(k^2)$ as $k \to \infty$. In section 4.1.2, we will demonstrate the asymptotic scaling

[4] Tabulated $\beta_i$ coefficients for the 4th-kind Chebyshev polynomials are available in table A.1.

of $\gamma^{-1}$ for the 1st-kind Chebyshev polynomial smoothers.

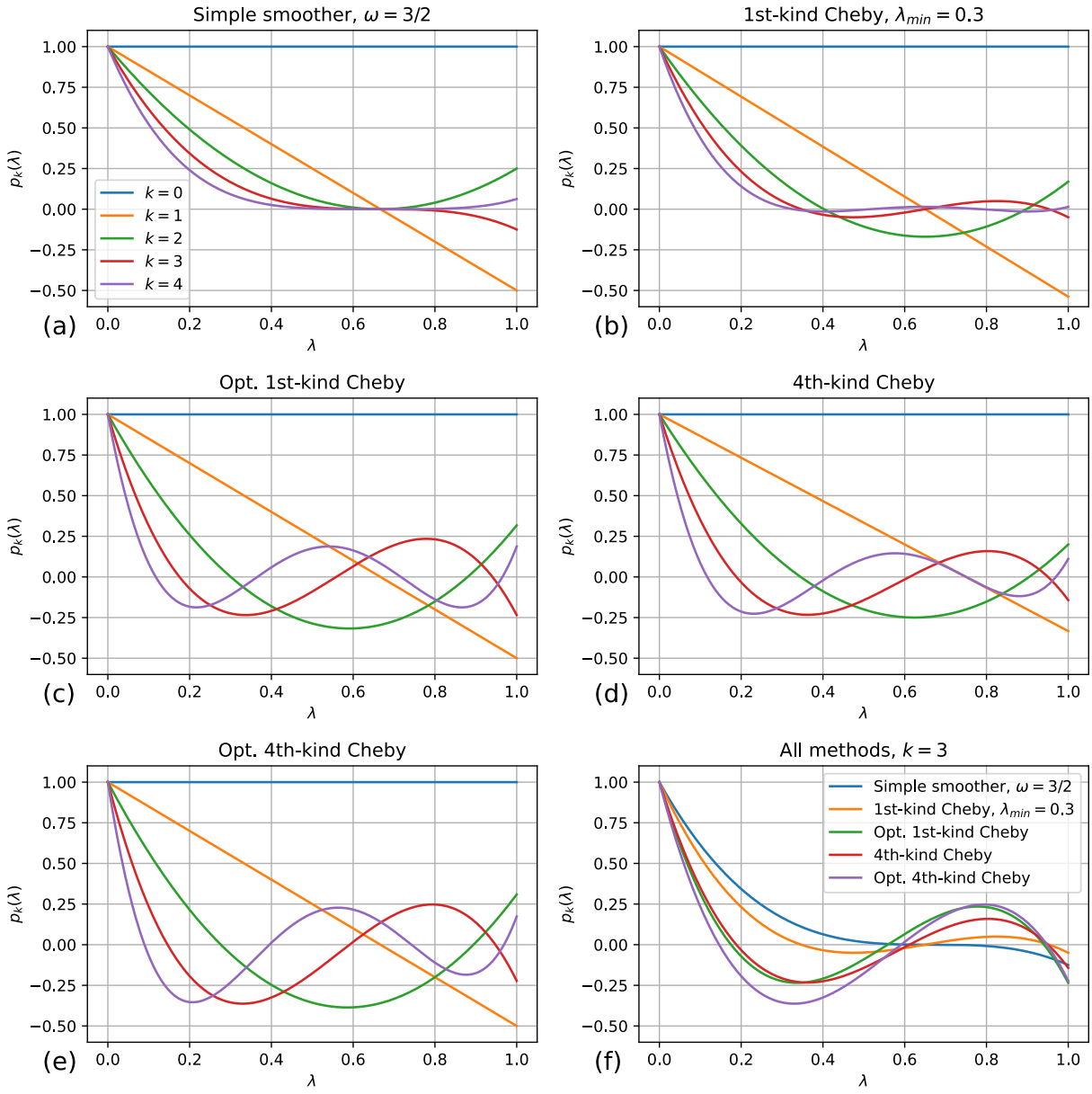



Figure 4.1: $p_k(\lambda)$ for various polynomial smoothers with varying polynomial orders, $k$. fig. 4.1a simple polynomial smoothing with $p_k(\lambda) = (1 - 3/2\lambda)^k$. fig. 4.1b smoothing with Chebyshev polynomials of the 1st-kind with $\lambda_{min} = 0.3$. fig. 4.1c smoothing with Chebyshev polynomials of the 1st-kind, $\lambda_{min}$ chosen to minimize eq. (4.7). fig. 4.1d smoothing with Chebyshev polynomials of the 4th-kind, $\beta_i = 1$. fig. 4.1e smoothing with Chebyshev polynomials of the 4th-kind, $\beta_i$ chosen to minimize eq. (4.7). fig. 4.1f all polynomials considered, with $k = 3$.

The method described in alg. 4.1 requires the same amount of computation/communi-

cation as the 1st-kind Chebyshev smoother in alg. 2.12 without needing the *ad-hoc* $\lambda_{min}$ parameter required by the 1st-kind Chebyshev smoother. As discussed later in section 4.1.2, the 4th-kind Chebyshev smoother converges *faster* than the 1st-kind Chebyshev smoother for the same amount of work, especially as the smoother order, $k$, increases. Further, alg. 4.1 is widely applicable in various geometric and algebraic multigrid method solvers. Therefore, as part of this dissertation, we have implemented alg. 4.1 in several popular open-source software packages, including `nekRS` [4], `petsc` [18], and `Trilinos/MueLu` [19], in order to make this method widely available. A further implementation of alg. 4.1 is available in `hypre` [81] from the following pull request: `https://github.com/hypre-space/hypre/pull/856`.

In subsequent sections, such as section 4.1.2 and section 4.2, it is helpful to have a concrete mechanism to estimate the multigrid approximation property constant, $C_j$, as defined in eq. (4.2). Following the interpretation presented by Lottes in [23], the multigrid approximation property constant in eq. (4.2) may be viewed as the condition number of $S_j A_j$ "restricted" to the $A_j$-orthogonal complement of the coarse-grid space, which is denoted

$$\pi_j = \left( I - P_{j+1}^j A_{j+1}^{-1} \left( P_{j+1}^j \right)^T A_j \right). \tag{4.12}$$

Without loss of generality, $S_j$ may be rescaled such that $\rho(S_j A_j) = 1$. Given that, the computation of multigrid approximation property constant, or the "restricted" condition number of $S_j A_j$, may be found by performing Lanczos iteration on $(S_j A_j)^{-1}$ wherein the candidate eigenvector is projected onto the $A_j$-orthogonal complement of the coarse-grid space using the coarse-grid correction operator, $\pi_j$, eq. (4.12). Algorithm 4.2 provides an algorithm to estimate the multigrid approximation property constant on a given multigrid level $j$.

### 4.1.2 Optimally Distributing Smoother Passes in the V-Cycle

Given $2k$ smoothing steps, what is the optimal way to distribute the number of smoothing steps for the pre-smoother, $m$, and the post-smoother, $n$, with $m + n = 2k$? The *multi-level* error bound in lemma 4.1, developed by Lottes in [23], provides a theoretical framework to answer the question.

Let us restrict the order considered in lemma 4.1 to the case $k_j = k$ for all $j \in 0, \ldots, \ell - 1$, then $\gamma_j = \gamma$ is constant across all levels. Further, let us define

$$C := \max_{j \in 0, \ldots, \ell - 1} C_j. \tag{4.13}$$

**Algorithm 4.2:** Lanczos estimate of multigrid approximation property

**Data:** Operator $A_j$, Prolongator $P_{j+1}^j$, Coarse Operator $A_{j+1}$, $m$ Lanczos iterations
**Result:** An estimate of the multigrid approximation property constant, $C_j$

$\underline{q}_1 = \pi_j \,\mathrm{rand}$, $\underline{q}_1 = \underline{q}_1 / \left\| \underline{q}_1 \right\|_2$
$\underline{r} = \pi_j \rho(S_j A_j) \left(S_j A_j\right)^{-1} \underline{q}_1$
$\alpha_1 = \underline{q}_1^T \underline{r}$
$\beta_1 = \left\| \underline{r} \right\|_2$
**for** $k = 2, \ldots, m$ **do**
  $\underline{q}_k = \underline{r} / \beta_{k-1}$
  $\underline{r} = \pi_j \left( \rho(S_j A_j) \left(S_j A_j\right)^{-1} \underline{q}_k - \beta_{k-1} \underline{q}_{k-1} \right)$
  $\alpha_k = \underline{q}_k^T \underline{r}$
  $\beta_k = \left\| \underline{r} \right\|_2$
**end**

$$C_j \approx \rho \left( \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_{m-1} \\ & & & \beta_{m-1} & \alpha_m \end{bmatrix} \right)$$

With eq. (4.13), lemma 4.1 simplifies to

$$\begin{aligned} \left\| E_{\searrow} \right\|_A^2 &\leq \frac{C}{C + \gamma^{-1}(k)} \\ &:= V(C,k). \end{aligned} \tag{4.14}$$

Rather than directly considering the effect of the choice of $m$ and $n$ on the multigrid error contraction factor, we instead consider the effect on the error contraction bound presented in eq. (4.14). The problem of interest, therefore, is to find $m^*, n^*$

$$\begin{aligned} m^*, n^* &:= \underset{m,n,m+n=2k}{\arg\min} \sqrt{V(C,m)} \cdot \sqrt{V(C,n)} \\ &= \underset{m,n,m+n=2k}{\arg\max} \, C \left( \gamma^{-1}(m) + \gamma^{-1}(n) \right) + \gamma^{-1}(m) \cdot \gamma^{-1}(n), \end{aligned} \tag{4.15}$$

where $\gamma^{-1}(k)$ is the inverse of $\gamma$, as defined in eq. (4.7), for a $k$-order polynomial.

While Lottes [23] provides asymptotic results for $\gamma^{-1}(k)$ for the 4th-kind and optimized 4th-kind Chebyshev polynomial smoothers, we provide the same information here for the 1st-kind Chebyshev polynomial smoother, both with fixed $\lambda_{min}$ and $\lambda_{min}$ values chosen to

optimize eq. (4.7). This is required in order to apply the optimization problem posed in eq. (4.15) to the 1st-kind Chebyshev polynomial smoother. To apply lemma 4.1 to the 1th-kind Chebyshev polynomial, without loss of generality, let us assume that $\lambda_{max} = 1$. For the 1st-kind Chebyshev polynomials with fixed $\lambda_{min}$,

$$\gamma_1^{-1}(k) = \begin{cases} \left(T_k\left(\frac{\lambda_{min}+1}{\lambda_{min}-1}\right)\right)^2 - 1 & k \leq k^* \\ \\ \frac{4kU_{k-1}\left(\frac{\lambda_{min}+1}{\lambda_{min}-1}\right)}{(\lambda_{min}-1)T_k\left(\frac{\lambda_{min}+1}{\lambda_{min}-1}\right)} & k > k^* \end{cases}, \tag{4.16}$$

where $T_\xi$ and $U_\xi$ are the $\xi$th-order Chebyshev polynomials of the first and second kinds, respectively. For $\lambda_{min} = 0.1$, $k^* = 3$. As $k \to \infty$, eq. (4.16) scales as

$$\gamma_1^{-1}(k) \sim 2\sqrt{\frac{1}{\lambda_{min}}}k. \tag{4.17}$$

However, by choosing $\lambda_{min}$ such that $\gamma^{-1}$ is maximized,

$$\gamma_{1_{opt}}^{-1}(k) \sim 2.38k^{1.73} \tag{4.18}$$

for large $k$. As an aside, a correlation for $\lambda_{min}^*$ with 1% relative error and 0.1% absolute error for $k \in [1, 50]$ is given by

$$\lambda_{min}^* \approx \frac{1.69}{k^{1.68} + 2.11k + 1.98}. \tag{4.19}$$

Asymptotic scaling for $\gamma^{-1}$ as $k \to \infty$ are summarized for the various polynomial smoothers in table 4.1.

Due to symmetry of eq. (4.15), the error bound for $(m, n)$ is the same as that of $(n, m)$. To start to assess eq. (4.15), we first consider the case with $m = n = k$ compared to $m = 2k$, $n = 0$. For the simple smoother with fixed $\omega$, the error-bound is minimized with $m = n$. For the 4th-kind Chebyshev polynomial, $m = 2k$, $n = 0$ outperforms the symmetric $m = n = k$ if and only if

$$C > \frac{2\left(k+1\right)^2}{3}. \tag{4.20}$$

The optimized 4th-kind Chebyshev polynomial has a similar condition, provided

$$C > \frac{2\left(6\left(2k+1\right)^2 - \pi^2\right)^2}{3\pi^2\left(-12\left(2k+1\right)^2 + 6\left(4k+1\right)^2 + \pi^2\right)} \tag{4.21}$$

Table 4.1: Asymptotic scaling of $\gamma^{-1}$ for various polynomial smoothers in the limit as $k \to \infty$. The effect of the polynomial smoother on the multigrid error bound in lemma 4.1 is $\gamma^{-1}$. Since this term appears in the denominator, a larger $\gamma^{-1}$ implies a smaller error contraction rate, which is desirable. The scaling for the simple multi-sweep, 4th-kind Chebyshev, and 4th-kind optimal Chebyshev smoothers are provided by Lottes in [23].

| Polynomial Smoother | $\gamma^{-1}, k \to \infty$ |
|---|---|
| Simple multi-sweep, damping | $2\omega k$ |
| 1st-kind Chebyshev, *fixed* $\lambda_{min}$ | $2\sqrt{\frac{1}{\lambda_{min}}}k$ |
| 1st-kind Chebyshev, $\lambda^*_{min}$ optimizes $\gamma^{-1}$ | $2.38k^{1.78}$ |
| 4th-kind Chebyshev | $\frac{4}{3}k(k+1)$ |
| 4th-kind optimal Chebyshev | $\frac{4}{\pi^2}(2k+1)^2 - \frac{2}{3}$ |

For the 1st-kind Chebyshev polynomial with fixed $\lambda_{min}$,

$$C \gtrapprox 1.55e^{1.45k} \tag{4.22}$$

as $k \to \infty$. However, for $k \in [1, 3]$, $m = 2k$, $n = 0$ outperforms $m = n = k$, irrespective of $C$. Lastly, for the 1st-kind Chebyshev polynomial with $\lambda_{min}$ chosen to maximize $\gamma^{-1}$,

$$C \gtrapprox 1.81k^{1.73}. \tag{4.23}$$

The critical $C$ values for the various polynomial smoothers are summarized in table 4.2. As we will demonstrate later in section 4.2, the critical $C$ value tabulated from lemma 4.1 in table 4.2 accurately predicts the region where the $m = 2k$, $n = 0$ scheme outperforms the $m = n = k$ scheme. This is the case for both *multigrid as a solver*, as considered in lemma 4.1, and *multigrid as a preconditioner*.

With the regions where $m = 2k$, $n = 0$ outperforms $m = n = k$ determined, the authors wish to solve the more general optimization problem in eq. (4.15). In lieu of a general solution, however, the authors opt to prove that the optimal $m$ and $n$ occurs at either $m = 2k$, $n = 0$ or $m = n = k$ for all $C > 0$. To do so, the authors utilized `sympy` [82] to solve for the region where a tentative $(\tilde{m}, \tilde{n})$ outperforms $m = 2k$, $n = 0$ and the region where $(\tilde{m}, \tilde{n})$ outperforms $m = n = k$. The intersection of these two regions, therefore, is the region where $(\tilde{m}, \tilde{n})$ outperforms both $m = 2k$, $n = 0$ and $m = n = k$. If the intersection of these two regions is empty for all $C > 0$, then the optimal solution $(m^*, n^*)$

Table 4.2: Given $2k$ smoothing passes, the critical $C$ value is the minimum value for which the $(2k, 0)$ scheme yields better convergence than the $(k, k)$ scheme as predicted in lemma 4.1.

| Polynomial Smoother | When to *omit* post smoothing? |
|---|---|
| Simple multi-sweep, damping | $C > \frac{(4k - \log(4k))^2}{\log(2k)}$ |
| 1st-kind Chebyshev, *fixed* $\lambda_{min} = 0.1$ | $k > 3$, $C \gtrapprox 1.55e^{1.45k}$ |
| 1st-kind Chebyshev, $\lambda^*_{min}$ optimizes $\gamma^{-1}$ | $C \gtrapprox 2.38k^{1.78}$ |
| 4th-kind Chebyshev | $C > \frac{2(k+1)^2}{3}$ |
| 4th-kind optimal Chebyshev | $C > \frac{2\left(6(2k+1)^2 - \pi^2\right)^2}{3\pi^2\left(-12(2k+1)^2 + 6(4k+1)^2 + \pi^2\right)}$ |

to eq. (4.15) must be either $(m^* = 2k, n^* = 0)$ or $(m^* = n^* = k)$ for all $C > 0$. Source code demonstrating this analysis for the fourth-kind Chebyshev polynomial smoother is given in fig. 4.2. With exception to $(4, 2)$ and $(5, 1)$ for $C < 4$ in the case of 1st-kind Chebyshev polynomial smoothing with $\lambda_{min} = 0.1$, the authors found that the optimal $(m^*, n^*)$ is either $(m^* = 2k, n^* = 0)$ or $(m^* = n^* = k)$ for all $C > 0$ for the smoothers considered, up to $k = 50$.

How does the results that the optimal smoothing passes $(m^*, n^*)$ is either $(m^* = 2k, n^* = 0)$ or $(m^* = n^* = k)$ effect the design of our multigrid preconditioner? First, the estimation of $C_j$ for each multigrid level as outlined in alg. 4.2 is no longer required to estimate whether the $(k, k)$ or $(2k, 0)$ scheme is optimal *for that given level*. A high-quality estimate of $C_j$ is sufficient to determine whether to employ $(k, k)$ or $(2k, 0)$ smoothing based on table 4.2, as demonstrated later in section 4.2. The authors note, however, that evaluating whether the $(k, k)$ or $(2k, 0)$ scheme converges faster only requires running two solves, one for each scheme, and comparing the convergence rates. This effort, which is amortized away as a one-time setup cost, is negligible compared to the inverse Lanczos method required in alg. 4.2, which requires several solves for the smoothed operator, $S_j A_j$, at each level.

A natural next question is on the expected improvement using the $(2k, 0)$ scheme over the symmetric $(k, k)$ scheme. This is done by taking the ratio of the error bound eq. (4.15) with $(k, k)$ and $(2k, 0)$ for the various polynomial smoothers considered. For the 1st-kind Chebyshev polynomial with $\lambda_{min} = 0.1$, the expected improvement in the multigrid convergence is no more than 6% for $k = 1$, and quickly decreases for larger $k$. Optimizing the bound with respect to $\lambda_{min}$, however, the 1st-kind Chebyshev polynomial with $(2k, 0)$ can outperform the symmetric $(k, k)$ scheme by 9%. For both the 4th-kind and optimized 4th-kind Cheby-

```
import sympy

def checkSmootherOptimality(kmax, obj):
  # m+n=2k: check optimal m^*\in {k,2k}, n^* \in {0,k} for all k \in [1,kmax]
  C = sympy.Symbol('C', positive=True, Real=True)
  posReals = sympy.Interval(0, sympy.oo)
  postSmoothOptimality = True
  for k in range(1,kmax+1):
    for m in range(k+1, 2*k):
      n = 2*k - m
      S_sym = sympy.solveset(obj(C,k,k) <= obj(C,m,n), C, domain=posReals)
      S_asym= sympy.solveset(obj(C,2*k,0) <= obj(C,m,n), C, domain=posReals)
      postSmoothOptimality &= (S_sym.intersect(S_asym) == sympy.EmptySet)
  return postSmoothOptimality

gammaFourth = lambda k : 4 * k * (k+1) / 3
objective = lambda C,m,n : C * (gammaFourth(m) + gammaFourth(n)) +
    gammaFourth(m) * gammaFourth(n)
assert checkSmootherOptimality(kmax=50, obj=objective)
```

Figure 4.2: Python code utilizing `sympy` to check the optimality of the $(2k, 0)$ and $(k, k)$ schemes.

shev polynomials, however, the $(2k, 0)$ scheme outperforms the symmetric $(k, k)$ scheme by 15% as $k \to \infty$. A numerical example demonstrating the applicability of this analysis based on lemma 4.1 and eq. (4.15) is given in section 4.2.

## 4.2 FINITE DIFFERENCE WITH GEOMETRIC MULTIGRID POISSON

In order to better appreciate the results of section 4.1, let us consider a simple $d = 2$ finite difference Poisson problem, eq. (1.1). The Poisson equation eq. (1.1) is considered with $d = 2$. Let $\Omega := [0, L_x] \times [0, L_y]$ be the domain of interest, with the boundary condition $u|_{\partial\Omega} = 0$. A finite difference grid of $(n + 1) \times (n + 1)$ points is considered, $n = 128$. For the purposes of this study, $u(x, y) = \sin(3\pi x/L_x) \sin(4\pi y/L_y) + g$, where $g$ is the same random vector with $g|_{\partial\Omega} = 0$. $L_x \geq 1$ is varied, while $L_y$ is fixed at unity. A geometric multigrid V-cycle with Chebyshev-accelerated Jacobi smoothing is employed as a preconditioner for GMRES(20). For the comparison with the bounds presented in lemma 4.1, multigrid is also considered as a solver. Each coarser level is discretized as $(n_c + 1) \times (n_c + 1)$, with $n_c = n/2$, as well as aggressive coarsening with $n_c = n/8$. This is repeated until there is only a single degree of freedom. As this case is meant to represent our target problem for

*unstructured* multigrid, semi-coarsening is not employed. On the coarsest level, the single degree of freedom system is solved exactly. We choose a relative residual reduction of $10^{-6}$ as the stopping criterion. A variety of smoothing orders are considered for all orders $m$ pre-smoothing and orders $n$ post-smoothing, with $m + n = 2k$ up to $k = 10$. For the $1^{st}$-Cheb, $\lambda_{min}^{opt}$, the $\lambda_{min}$ value is provided from the correlation from eq. (4.19). The multigrid approximation property constant is estimated using $m = 20$ Lanczos iterations, alg. 4.2.

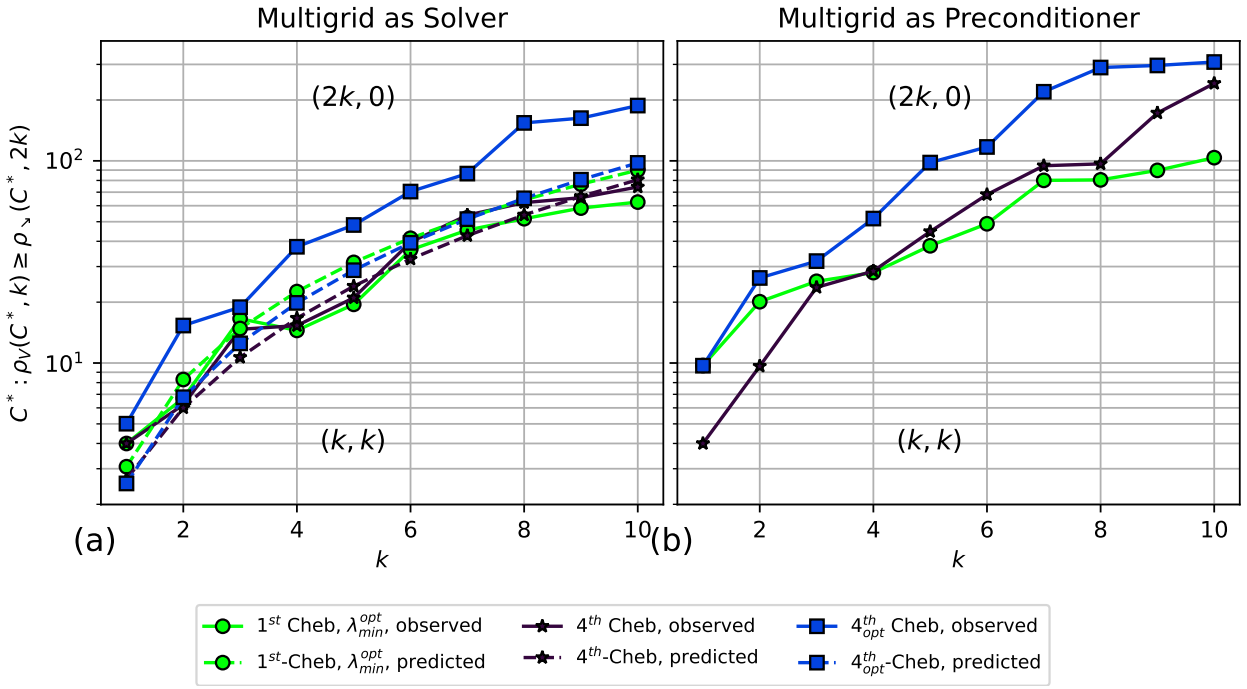



Figure 4.3: Critical $C^*$ at which $(2k, 0)$ converges faster than $(k, k)$ for $C > C^*$.

Convergence results are based on the observed average solver convergence rate

$$\rho = \exp\left(\frac{1}{N}\log\frac{\|\underline{r}_N\|_2}{\|\underline{r}_0\|_2}\right), \tag{4.24}$$

where $N$ is the number of iterations. The first question, as posed in the optimization problem eq. (4.15), is regarding the optimal choice of V-cycle smoothing $(m, n)$ with $m + n = 2k$. Recall, however, that the authors demonstrated the optimality of either the one-sided $(2k, 0)$ or symmetric $(k, k)$ V-cycles for all orders up to $k = 50$, with a few exceptions for small values of $C$. The authors observe that, as predicted in eqs. (4.14) and (4.15), the convergence rate of $(m, n)$ is nearly equivalent to $(n, m)$. Further, with few exceptions, either the one-sided $(2k, 0)$ or symmetric $(k, k)$ V-cycles yielded the smallest convergence rate, as defined in eq. (4.24).

With our consideration now on the use of the symmetric $(k, k)$ or one-sided $(2k, 0)$ V-

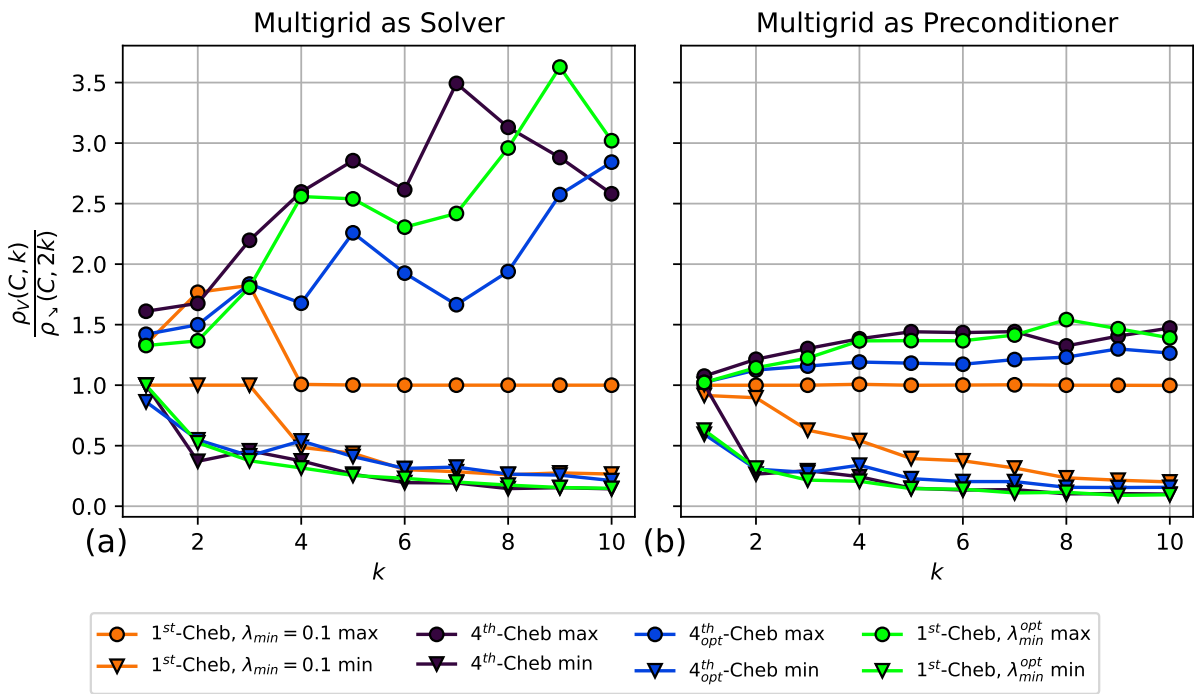



Figure 4.4: Max (min) error contraction rate ratios for $(2k, 0)$ and $(k, k)$ smoothing schemes.

cycles, we can now confirm our theoretical prediction of when to apply these methods, as outlined in eqs. (4.20) to (4.23). The convergence rate for the symmetric $(k, k)$ V-cycle is denoted $\rho_V(C, k)$, while the one-sided $(2k, 0)$ V-cycle is $\rho_\searrow(C, 2k)$. Values of $(C, k)$ at which $\rho_V(C, k) \geq \rho_\searrow(C, 2k)$ are shown in fig. 4.3. Predicted results are from eqs. (4.20) to (4.23), which are only applicable to multigrid as a *solver*, not *preconditioner*. Observed results are from 2D finite difference example. At a given $k$, $C > C^*$ indicates that the one-sided $(2k, 0)$ V-cycle yields a lower error bound than the symmetric $(k, k)$ V-cycle. Conversely, $C < C^*$, indicates that the symmetric V-cycle yields better convergence. For the 1st-kind with optimized $\lambda_{min}$ coefficient, 4th-kind, and optimized 4th-kind Chebyshev smoothers, the predicted domain in which to apply the one-sided $(2k, 0)$ V-cycle over the symmetric $(k, k)$ V-cycle shows agreement with the results obtained by experiment using multigrid as a solver (fig. 4.3a). As noted by eq. (4.22), when $k > 3$, the 1st-kind Chebyshev smoother does not benefit from applying the one-sided $(2k, 0)$ V-cycle over the symmetric $(k, k)$ V-cycle. At the same time, however, for $k \leq 3$, the one-sided $(2k, 0)$ V-cycle outperforms the symmetric $(k, k)$ V-cycle, irrespective of $C$. Despite the applicability of lemma 4.1 being limited to multigrid as a solver, the predicted domain in which to apply the one-sided $(2k, 0)$ V-cycle *as a preconditioner* is similar to that of using multigrid *as a solver* (fig. 4.3b).

The maximum and minimum ratio of the error contraction rates, along with the predicted performance, are shown in fig. 4.4. While the predicted performance benefit of applying

the one-sided V-cycle approach is limited, nevertheless, fig. 4.4a and fig. 4.4b demonstrate that the one-sided V-cycle offers an improvement compared to the symmetric V-cycle for problems with moderate values of $C$. However, when applied to relatively easy problems ($C \approx 1$), the one-sided V-cycle is a poor choice.

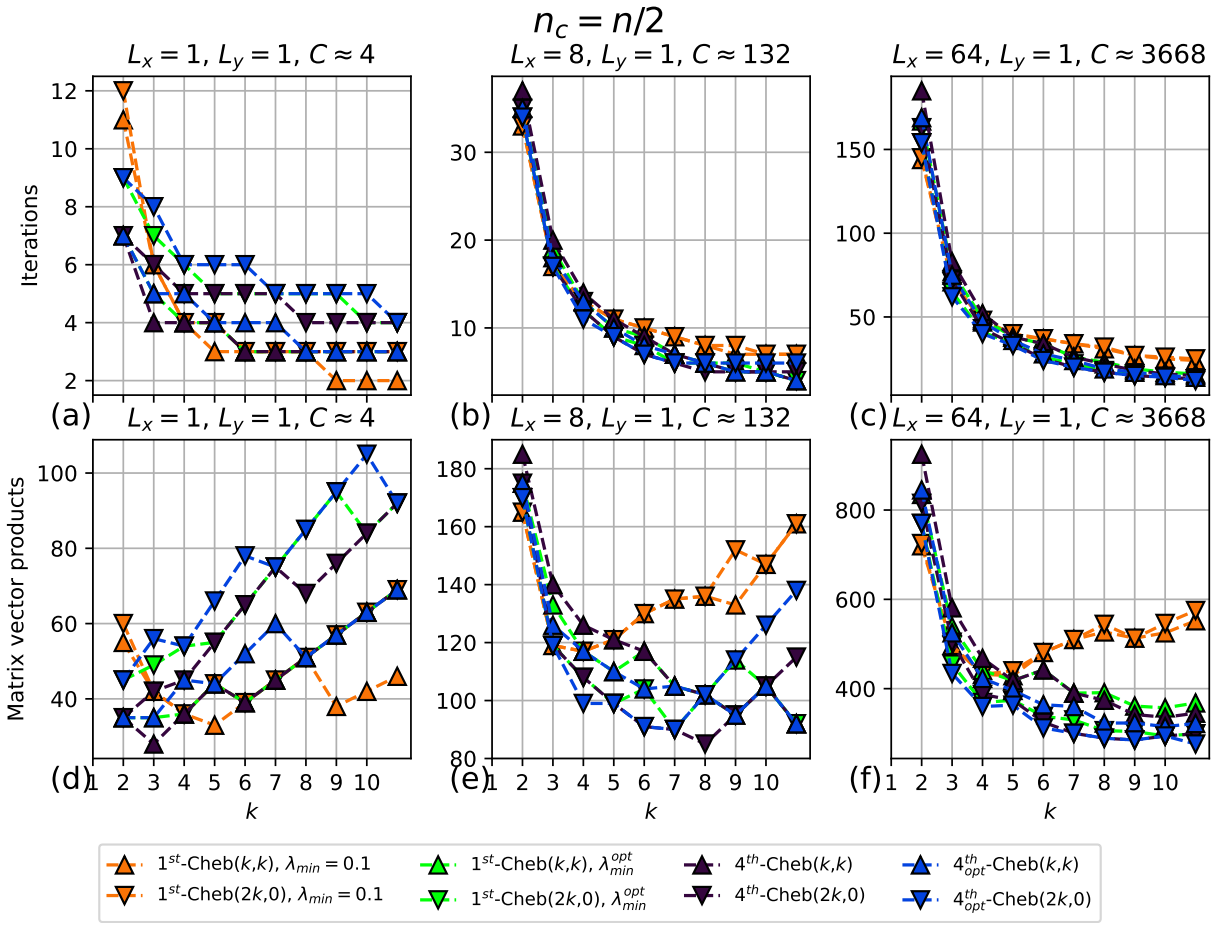



Figure 4.5: FD, $n_c = n/2$. Multigrid as preconditioner for GMRES(20).

Results for $n_c = n/2$ are shown in fig. 4.5, $n_c = n/8$ in fig. 4.6. In both, multigrid is used to precondition the GMRES(20) solver. In the case $n_c = n/2$ and $L_x = 1$ ($C \approx 4$), shown in fig. 4.5a,d, the work required, as measured by fine-grid matrix vector products, is minimized for relatively low orders. Further, applying the bounds shown in fig. 4.3a, this is the scenario in which the one-sided V-cycle approach *should not* be utilized. However, even at moderate grid aspect ratios, such as $L_x = 8$, $C \approx 132$ becomes large enough to justify the usage of the one-sided V-cycle approach, especially for the 4th-kind Chebyshev smoothers (fig. 4.5b,e) at moderate orders $k = 6$, or 12th-order for the $(2k, 0)$ case. This effect becomes even more apparent when $L_x = 64$ ($C \approx 3668$), shown in fig. 4.5c,f. In the results for $n_c = n/2$, the 1st-kind with optimized $\lambda_{min}$, 4th-kind, and optimized 4th-kind Chebyshev

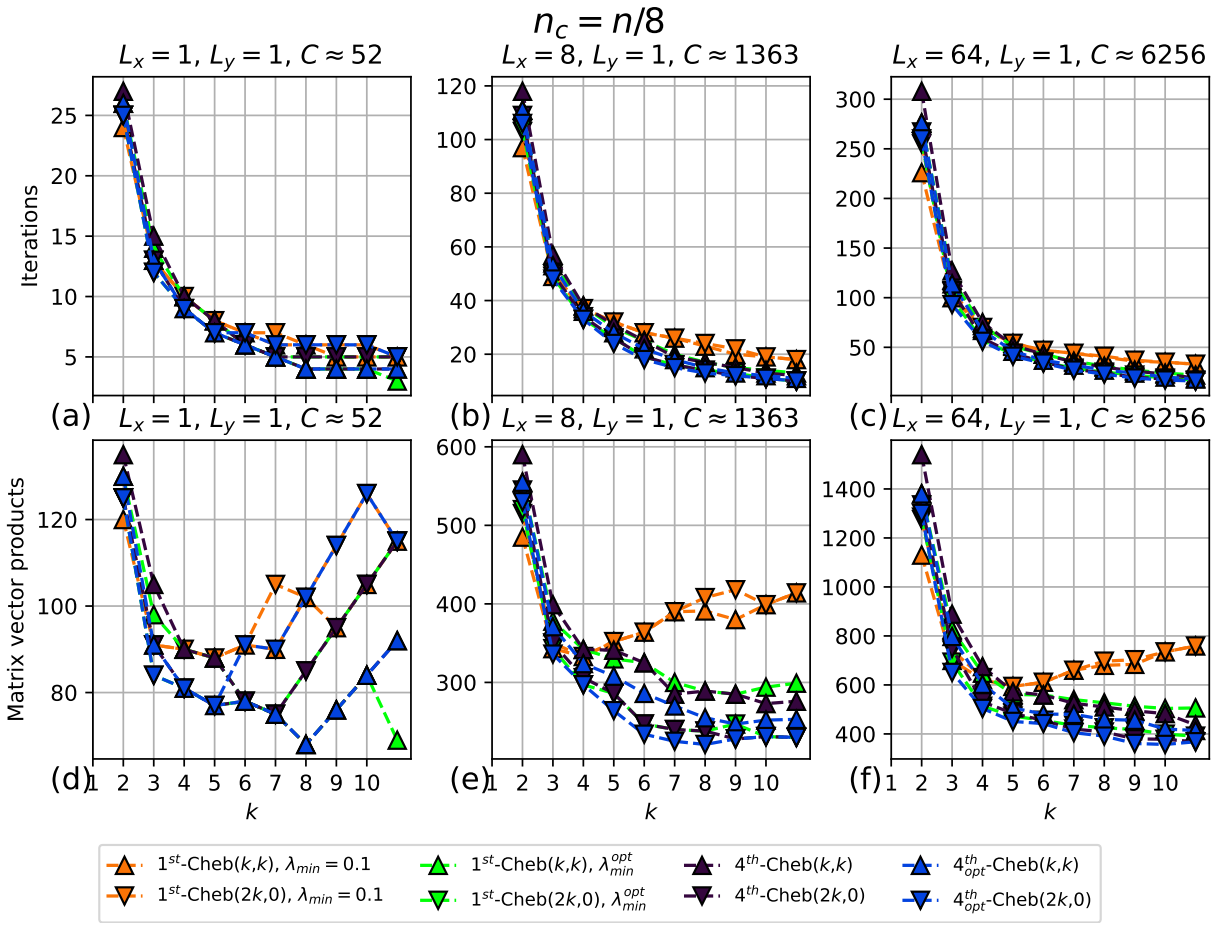



Figure 4.6: FD, $n_c = n/8$. Multigrid as preconditioner for GMRES(20).

smoothers greatly outperform the 1st-kind Chebyshev smoother, *especially* at high orders. Further, it is observed that both the iteration count and *total work* in matrix-vector products is minimized at high orders for the 1st-kind with optimized $\lambda_{min}$, 4th-kind, and optimized 4th-kind Chebyshev smoothers. This is not the case for the 1st-kind Chebyshev smoother, however. This effect of lowering both the iteration count and work has the additional benefit of reducing the number of coarse grid solves required for each iteration, whose cost is not factored in this analysis.

In the aggressive coarsening case ($n_c = n/8$), both the number of matrix-vector products and iterations are reduced through using higher-order Chebyshev smoothing irrespective of the grid aspect ratio for all but the standard 1st-kind Chebyshev smoothers. Secondly, the one-sided $(2k, 0)$ V-cycle approaches generally outperforms the full, symmetric $(k, k)$ V-cycle approach. There are two important implications from this result: first, there exists considerable benefit in transitioning a multigrid solver to either construct high-quality estimates for $\lambda_{min}^{opt}$ for the 1st-kind Chebyshev smoother, such as those provided in correlation eq. (4.19),

or utilize one of the 4th-kind Chebyshev smoothers; and second, additional performance can be achieved by using the one-sided $(2k, 0)$ V-cycle approach at the expense of symmetry, the implication of which is further discussed in section 2.2.4.

Table 4.3: Multigrid hierarchy for 2D finite differences, $n_c = n/2$

| Level | Rows | nnz$(\cdot)$ | nnz$(\cdot)/row$ | Sparsity |
|---|---|---|---|---|
| 1 | 16129 | 80137 | 4.968 | 0.999 |
| 2 | 3969 | 34969 | 8.810 | 0.997 |
| 3 | 961 | 8281 | 8.617 | 0.991 |
| 4 | 225 | 1849 | 8.217 | 0.963 |
| 5 | 49 | 361 | 7.367 | 0.849 |
| 6 | 9 | 49 | 5.444 | 0.395 |
| 7 | 1 | 1 | 1 | 0 |

Table 4.4: Multigrid hierarchy for 2D finite differences, $n_c = n/8$

| Level | Rows | nnz$(\cdot)$ | nnz$(\cdot)/row$ | Sparsity |
|---|---|---|---|---|
| 1 | 16129 | 80137 | 4.968 | 0.999 |
| 2 | 225 | 1849 | 8.217 | 0.963 |
| 3 | 1 | 1 | 1 | 0 |

Let us also consider the trade-offs associated with using $n_c = n/2$ and $n_c = n/8$ as coarsening strategies. The multigrid hierarchies for $n_c = n/2$ and $n_c = n/8$ are shown in table 4.3 and table 4.4, respectively. Despite improvements in the convergence rate from using a different V-cycle approach or Chebyshev smoother, the number of matrix-vector products required by the solve is greater for the aggressive coarsening case. However, the grid complexity

$$\frac{\sum_{j=0}^{\ell} \text{nnz}(A_j)}{\text{nnz}(A_0)} \tag{4.25}$$

for $n_c = n/2$ is 1.568, while for $n_c = n/8$ it is only 1.023. Therefore, the *amount* of work done per cycle is less for the aggressive coarsening case, even ignoring the cost of the grid-transfer operators. An additional benefit of the aggressive coarsening strategy is decreasing the number of multigrid levels from 7 with $n_c = n/2$ to 3 with $n_c = n/8$. This feature is especially attractive in a parallel computing context where each additional multigrid level requires additional communication cost, as noted in chapter 3. Significant effort has been spent on improving the parallel scalability of multigrid methods, especially in the AMG context, see [83, 84, 85]. One strategy to achieve better scalability is to rely on aggressive coarsening

strategies, which require more robust smoothers, such as the Chebyshev smoothers discussed in section 4.1.

The results shown in fig. 4.5 and fig. 4.6 are summarized in table 4.5. The solver configuration yielding the lowest number of matrix-vector products is listed for each case. $4^{th}_{opt}$-Cheb, Jacobi(20,0), for example, denotes a 20th order Chebyshev smoother of the optimized 4th-kind, using one-sided smoothing (thereby, having the same cost per iteration as order 10 with the symmetric V-cycle). This solver configuration yields the lowest number of matrix-vector products for $L_x = 128$ with aggressive coarsening. We see that, for high aspect ratio grids and aggressive coarsening, the one-sided $(2k, 0)$ V-cycle offers superior performance to the symmetric $(k, k)$ V-cycle approach.

Table 4.5: Solver configuration with lowest number of matrix-vector products for finite different geometric multigrid

| $L_x$ | $n_c = n/*$ | Complexity | Solver | Mat-Vec | Iterations | Work |
|---|---|---|---|---|---|---|
| 1 | 2 | 1.568 | $4^{th}$-Cheb, Jacobi(2, 2) | 20 | 4 | 31.4 |
| 2 | 2 | 1.568 | $4^{th}$-Cheb, Jacobi(3, 3) | 28 | 4 | 43.9 |
| 4 | 2 | 1.568 | $4^{th}_{opt}$-Cheb, Jacobi(5, 5) | 44 | 4 | 69.0 |
| 8 | 2 | 1.568 | $4^{th}$-Cheb, Jacobi(14, 0) | 75 | 5 | 117.6 |
| 16 | 2 | 1.568 | $4^{th}$-Cheb, Jacobi(20, 0) | 126 | 6 | 197.6 |
| 32 | 2 | 1.568 | $4^{th}_{opt}$-Cheb, Jacobi(20, 0) | 189 | 9 | 296.4 |
| 64 | 2 | 1.568 | $4^{th}_{opt}$-Cheb, Jacobi(20, 0) | 252 | 12 | 395.1 |
| 128 | 2 | 1.568 | $4^{th}_{opt}$-Cheb, Jacobi(20, 0) | 252 | 12 | 395.1 |
| 1 | 8 | 1.023 | $4^{th}$-Cheb, Jacobi(7, 7) | 60 | 4 | 61.4 |
| 2 | 8 | 1.023 | $4^{th}_{opt}$-Cheb, Jacobi(8, 0) | 81 | 9 | 82.9 |
| 4 | 8 | 1.023 | $4^{th}$-Cheb, Jacobi(20, 0) | 126 | 6 | 128.9 |
| 8 | 8 | 1.023 | $4^{th}_{opt}$-Cheb, Jacobi(14, 0) | 195 | 13 | 199.5 |
| 16 | 8 | 1.023 | $4^{th}_{opt}$-Cheb, Jacobi(18, 0) | 266 | 14 | 272.1 |
| 32 | 8 | 1.023 | $4^{th}_{opt}$-Cheb, Jacobi(20, 0) | 315 | 15 | 322.2 |
| 64 | 8 | 1.023 | $4^{th}_{opt}$-Cheb, Jacobi(18, 0) | 323 | 17 | 330.4 |
| 128 | 8 | 1.023 | $4^{th}_{opt}$-Cheb, Jacobi(20, 0) | 294 | 14 | 300.8 |

The results shown in fig. 4.3 demonstrate the applicability of lemma 4.1 in determining whether to employ the symmetric $(k, k)$ versus the $(2k, 0)$ V-cycle scheme. However, a natural related question is: *How well does lemma 4.1 actually predict the convergence rate of the multigrid method?* The convergence rate for the finite difference case with multigrid *as a solver* is shown as a solid line in figs. 4.7 and 4.8 for each polynomial smoother, along with the error bound from lemma 4.1 as the dashed line. Figure 4.8 contains the results for the symmetric $(k, k)$ V-cycle scheme, while fig. 4.7 contains those from the asymmetric

$(2k, 0)$ scheme. There are two important observations from figs. 4.7 and 4.8. The bound from lemma 4.1 proves both *useful*, in that it does not exceed unity and, as expected, serves as an upper bound to the observed convergence rate, as noted by [23]. However, lemma 4.1 is not *sharp*. The difference between the predicted and observed convergence rate is large. Fortunately, the relative ranking of each polynomial smoother is, however, preserved when comparing the predicted and observed convergence rate. For example, the bound in fig. 4.8c predicts that the relative ranking, from best to worst smoother polynomial, is: $4^{th}_{opt}$-Cheb, $4^{th}$-Cheb, $1^{st}$-Cheb, $\lambda^*_{min}$, and $1^{st}$-Cheb. This matches the observed convergence rates.

What if a more accurate predictor of the convergence rate is required, however? The Local Fourier Analysis (LFA) methods discussed in section 4.3 may provide more accurate estimates for the multigrid convergence rate. Two-level LFA predicted convergence rates from `LFAToolkit.jl` are shown as a dotted lines in figs. 4.7 and 4.8 for each polynomial smoother [20]. The LFA methods are able to predict the convergence rate more accurately than lemma 4.1, as shown by the smaller difference between the predicted and observed convergence rate. For example, the LFA-predicted convergence rate is nearly identical to the observed rate in fig. 4.7a. At the same time, however, the LFA prediction does not provide an upper-bound to the observed convergence rate. Figure 4.8b, for example, shows that often the LFA-predicted convergence rate is too optimistic. In other scenarios, the LFA-predicted convergence rate is nearly identical to those predicted by lemma 4.1, as shown in fig. 4.8d. As further discussed in section 4.3, the LFA-predicted convergence rate is for a two-level method. By employing LFA for the multilevel case, however, more accurate predictions of the convergence rate may be obtained. This, however, is omitted from the current study, as the computational expense of the multilevel LFA is *more* expensive than performing the solve itself when considering the aggressive coarsening case, $n_c = n/8$. This prevents the multilevel LFA from being employed as a predictor of the relative performance of each of the polynomial smoothers. While one could imagine forming optimal polynomials with respect to an LFA-predicted convergence rate as opposed to lemma 4.1, this is left as future work, see section 8.2.

## 4.3 LOCAL FOURIER ANALYSIS

The construction of the 4th-kind Chebyshev polynomial smoother presented in section 4.1 relied on solving a *weighted* mini-max problem based on a two-level error bound from Hackbusch, eq. (4.1)[79]. However, we observe that, through the work of Lottes in lemma 4.1, an *optimal* polynomial smoother could be constructed with respect to the error bound[23]. However, as observed toward the end of section 4.2, the convergence rate predicted by lemma 4.1

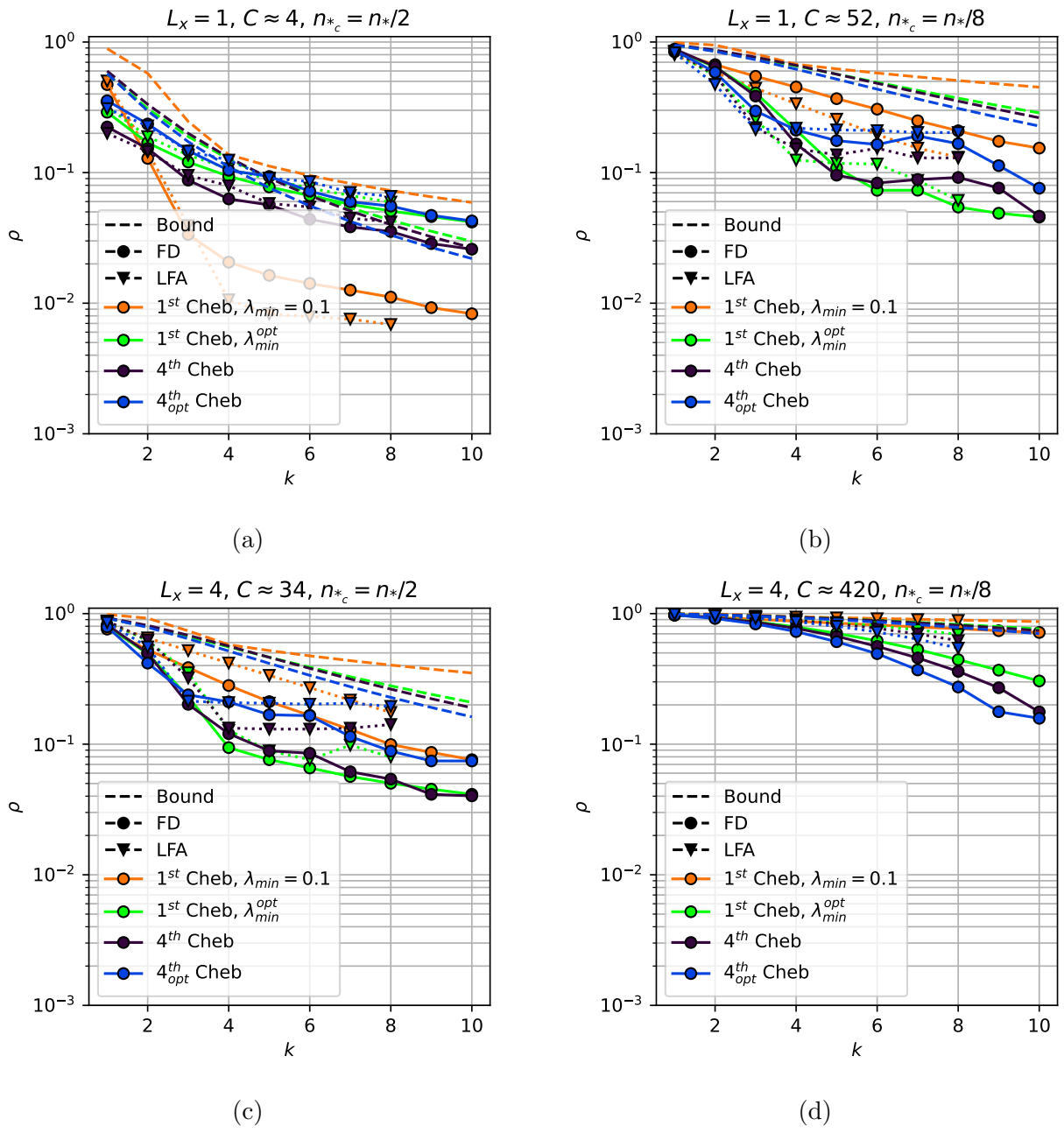



Figure 4.7: Finite-difference convergence factor for the $(2k, 0)$ V-cycle scheme using multigrid *as a solver* (solid line), compared with predicted convergence factor from lemma 4.1 (dashed line). Rows correspond to fixed $L_x$ values: $L_x = 1$ for figs. 4.7a and 4.7b and $L_x = 4$ for figs. 4.7c and 4.7d. Columns correspond to different coarsening factors: $n_c = n/2$ for figs. 4.7a and 4.7c and $n_c = n/8$ for figs. 4.7b and 4.7d.

is not *sharp*. Through the use of Local Fourier Analysis (LFA) [86, 87], sharper predictions on the convergence rate of polynomial smoothers can be made.

LFA is a method for analyzing the convergence rate of multigrid methods by considering

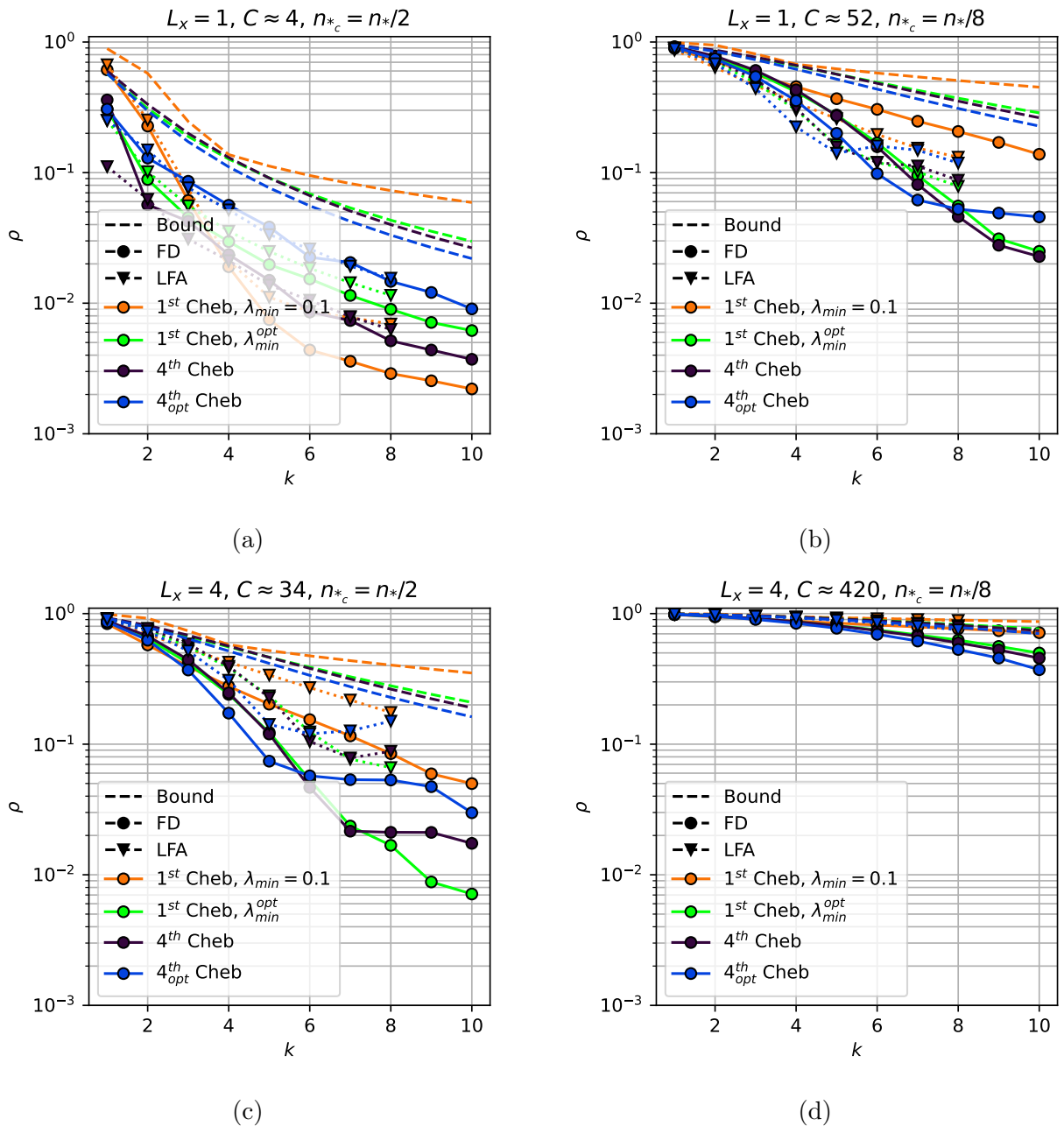



Figure 4.8: Finite-difference convergence factor for the symmetric $(k, k)$ V-cycle scheme using multigrid *as a solver* (solid line), compared with predicted convergence factor from lemma 4.1 (dashed line). Rows correspond to fixed $L_x$ values: $L_x = 1$ for figs. 4.8a and 4.8b and $L_x = 4$ for figs. 4.8c and 4.8d. Columns correspond to different coarsening factors: $n_c = n/2$ for figs. 4.8a and 4.8c and $n_c = n/8$ for figs. 4.8b and 4.8d.

the Toeplitz operators that arise from low-order finite element and finite difference discretizations on infinite uniform grids with grid point spacing $h$. Given the Toeplitz structure, the operators are diagonalized by the standard Fourier modes $\varphi(\theta, x) = e^{\imath\theta x/h}$, with $i^2 = -1$,

which are periodic with $\varphi(\theta + 2\pi, x) = \varphi(\theta, x)$ on all grid points $x \in h\mathbb{Z}$. This can be extended to high-order $p$-multigrid methods, as in [47], by extending this analysis to block-Toeplitz operators, such as those arising from the discretization presented in section 2.1. Once in the frequency domain, the eigenvalues of the *symbol* of the multigrid error propagator, $\tilde{E}$, which is the error propagator in the frequency domain, can be computed. For standard coarsening from a fine grid with a mesh size $h$ to a coarse grid with a mesh size of $H = 2h$, low frequencies are given by $\theta \in T^{\text{low}} = [-\pi/2, \pi/2)^d$ and high frequencies are given by $\theta \in T^{\text{high}} = [-\pi/2, 3\pi/2)^d \setminus T^{\text{low}}$. Once the eigenvalues of $\tilde{E}$ are known, the convergence rate of the multigrid method can be computed as

$$\max_{\theta} \rho(\tilde{E}). \tag{4.26}$$

For a more thorough introduction into LFA, the reader is referred to [47].

To better understand the smoother polynomials introduced in section 4.1, LFA is employed using the tools from [47], wherein the authors introduce `LFAToolkit.jl` [20], an automated tool for performing LFA for both $p$-type and $h$-type multigrid. `LFAToolkit.jl` is extended to support smoothing via the fourth and optimized fourth-kind Chebyshev polynomial smoothers from section 4.1. This is done by computing the *symbol* of eq. (4.3) for the fourth-kind Chebyshev polynomial smoother and the *symbol* of eq. (4.8). Following the analysis in [47], this simply becomes the polynomials from the aforementioned equations applied to the symbol of the smoother. As an aside, *both* the fourth-kind Chebyshev polynomial smoother and the optimized fourth-kind Chebyshev polynomial smoother are implemented in `LFAToolkit.jl` to enable users the ability to perform automated LFA for these polynomial smoothers.

$p$-multigrid results for a uniform aspect ratio grid and $4:1$ aspect ratio grid are presented in fig. 4.9 and fig. 4.10, respectively. These plots show the max eigenvalues of the symbol of the two-level error propagator. The resulting convergence rate is computed as eq. (4.26). The various polynomial smoothers considered in section 4.1 are compared for a two-level $p$-multigrid scheme with $p = 7$ and $p = 1$. As predicted by lemma 4.1, the fourth and optimized fourth-kind Chebyshev polynomials outperform the standard 1st-kind Chebyshev polynomial smoother, especially at high smoother orders.

Two-level $h$-multigrid LFA for the finite difference case in section 4.2 is applied for the two aspect ratios, $L_x = 1, 4$, and for the two coarsening factors considered, $n_c = n/2$ and $n_c = n/8$. This is repeated for the four kinds of polynomial smoothers considered in section 4.1 across the various polynomial orders for both the $(k, k)$ and $(2k, 0)$ cycle schemes. These LFA-predicted convergence rates are reported in section 4.2, specifically

figs. 4.7 and 4.8, and are shown to be generally sharper than the convergence rates predicted by lemma 4.1. The reason a two-level LFA bound is employed in section 4.2, as opposed to a multilevel bound, is that performing the multilevel LFA for the case of $n_c = n/8$ coarsening for the three-level V-cycle requires *more* computation in evaluating eq. (4.26) than solving the resultant system using the multigrid method. As this LFA is meant to *predict* the solver performance without requiring the expense of performing the actual solve, this is not a viable option. As such, we employ a two-level LFA for the results in figs. 4.7 and 4.8.

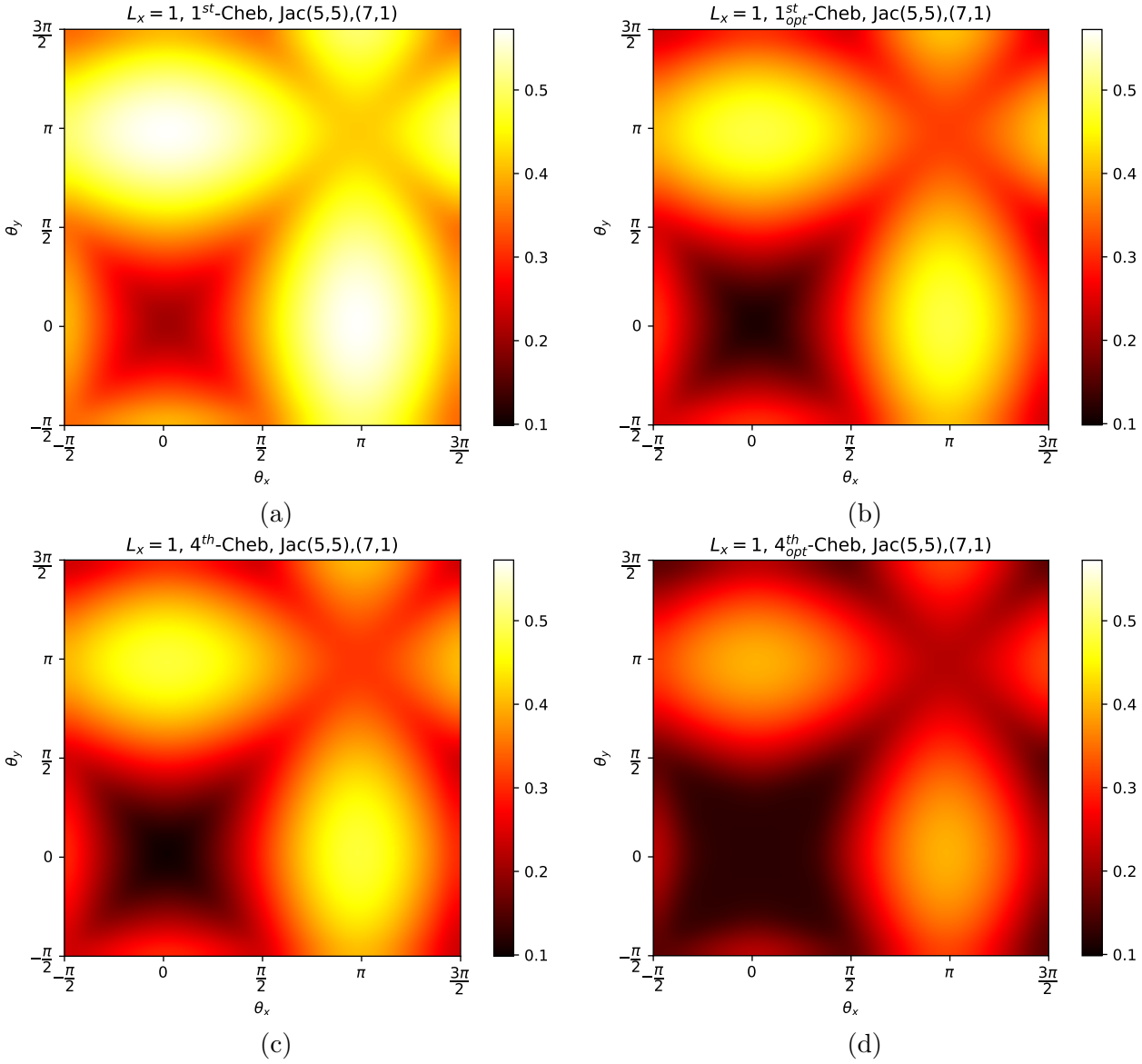
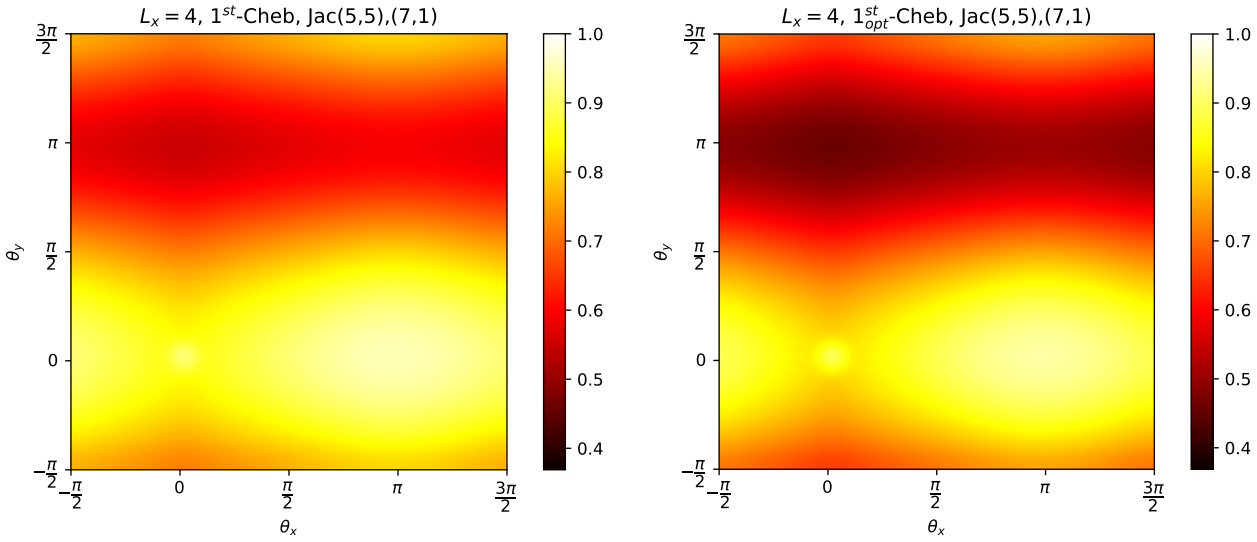
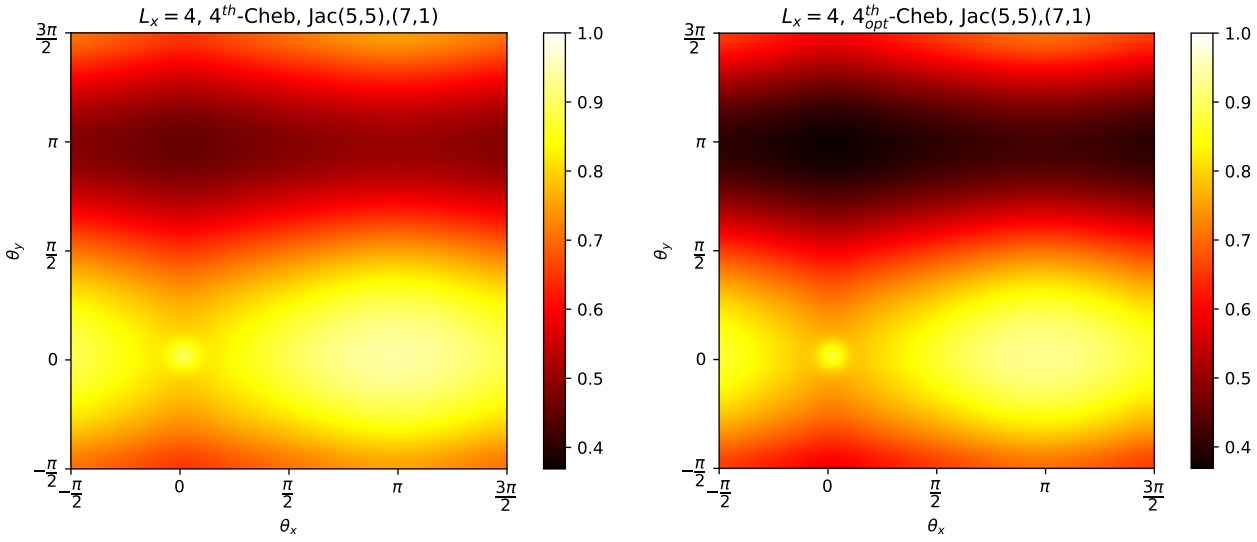



Figure 4.9: LFA results for a two-level scheme at $p = 7$ for various polynomial smoothers, all at order $k = 5$ with a symmetric $(k, k)$ cycle. Grid aspect ratio is $1 : 1$.

One could imagine, instead of finding the optimal polynomial with respect to lemma 4.1,




(a) (b)




(c) (d)

Figure 4.10: LFA results for a two-level scheme at $p = 7$ for various polynomial smoothers, all at order $k = 5$ with a symmetric $(k, k)$ cycle. Grid aspect ratio is $4 : 1$.

one could instead find the optimal polynomial with respect to the convergence rate predicted by LFA as in eq. (4.26). While this is still left as potential future work as discussed in section 8.2, the direct polynomial smoother optimization methods considered in section 4.5 prove promising.

## 4.4 NUMERICAL RESULTS

Here we consider the solver performance results for the test cases highlighted in section 2.8. Similar to section 2.9, the results presented here are on Summit, with 6 V100 GPUs and 42 IBM Power9 CPUs per node. Each rank is assigned to a single GPU. To limit the number of runs required, we consider only varying the Chebyshev polynomial smoother, smoother type, polynomial order, and the distribution of smoothing passes amongst the pre-/post-smoothing phases of the V-cycle. The strong scaling limits for each case, as identified in section 2.9.2, are used for the Navier-Stokes cases, which are reported in table 2.6.

### 4.4.1 Kershaw Results

Results for the Kershaw case for $\varepsilon = 1, 0.3, 0.05$ are shown in figs. 4.11 to 4.13. Note the mesh is generated with $E = 36^3$ elements with a polynomial order $p = 7$. A single Summit note with $P = 6$ V100 GPUs is used for all Kershaw results, putting $n/P \approx 2.7M$ degrees-of-freedom per GPU. This represents the scalability limits identified in table 2.6. To mitigate the effects of system noise, the reported solve times are based on minimum time to solution over 50 trials. The reported number of matrix-vector products are for the *finest grid*, $p = 7$.

When $\varepsilon = 1$, the time to solution is minimized by utilizing a symmetric V-cycle with relatively low-order Chebyshev-accelerated RAS smoothing. For this case, SEMFEM is a comparatively poor method. While the iteration count is decreased with respect to increasing order, the overall cost of applying the heavier smoother translates to a higher cost per solve. This can especially be observed in the increased work requirement in terms of matrix-vector products, fig. 4.11d-f. At larger scales, however, the scalability of the AMG coarse grid solve may dominate the cost of the preconditioner, and thus anything to reduce the number of coarse grid solves may prove beneficial. In this scenario one should expect the multigrid approximation property constant eq. (4.2) to be quite low for the case with no geometric deformation. In this regime, the theoretical prediction in fig. 4.3, states that the convergence is improved using a symmetric $(k, k)$ V-cycle with as opposed to the one-sided $(2k, 0)$ V-cycle. Bench-marking demonstrates that a single boomerAMG V-cycle iteration for $p = 1$ on the CPU is nearly 12 times the cost of a matrix-vector product, see fig. 3.19 from section 3.3 for additional information. For larger cases, wherein the number of AMG levels needed for a single boomerAMG V-cycle increases, the relative cost of the coarse grid solve will increase relative to the cost of a matrix-vector product.

When $\varepsilon = 0.3$, however, higher-order Chebyshev smoothing can yield a lower time to

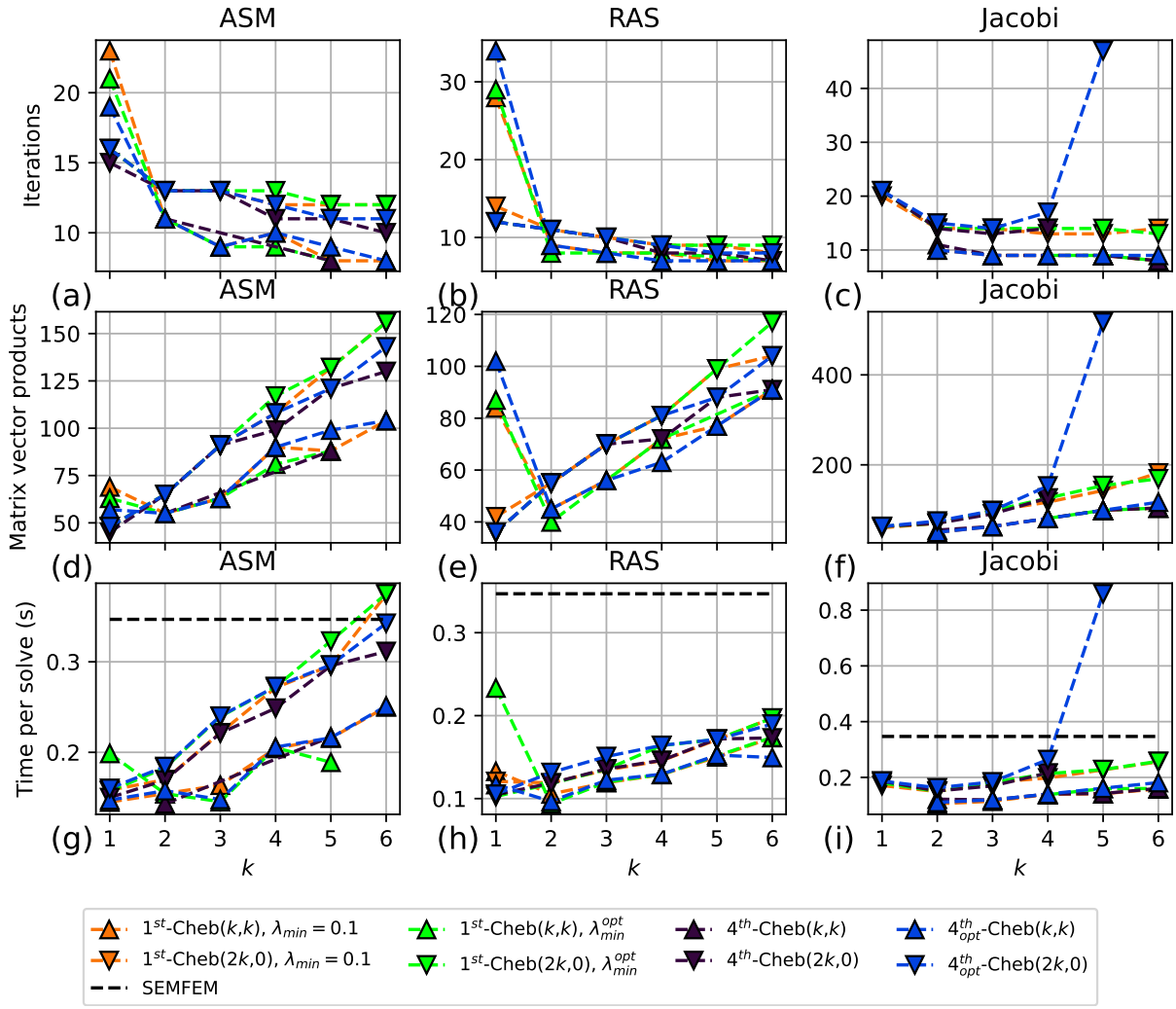



Figure 4.11: Kershaw results, $\varepsilon = 1$.

solution. As shown in fig. 4.12d, the time to solution for the $4^{th}$ and $4^{th}_{opt}$ Chebyshev schemes tend to improve with high orders, even up to $k = 6$ for the full V-cycle (or 12 for the half V-cycle). This demonstrates a major improvement over the standard first kind Chebyshev scheme, which tends to yield a minimum time to solution at $k = 3$. Remarkably, the 4th-kind and optimized 4th-kind Chebyshev schemes are generally equivalent to, if not better than, optimizing the $\lambda_{min}$ parameter for the standard first-kind Chebyshev scheme. This allows for increased performance with high order Chebyshev smoothing, without the requirement of tuning an additional parameter. [5] Similar to the $\varepsilon = 1$ case, SEMFEM is not the preconditioner yielding the fastest time to solution. However, this approach becomes comparable

[5] In large-scale fluid mechanics applications, this overhead is easily amortized over the $10^4$–$10^6$ time-steps required. Reasonably good values of $\lambda^{opt}_{min}$ do not depend on the RHS, allowing this to be part of the setup cost of the preconditioner.

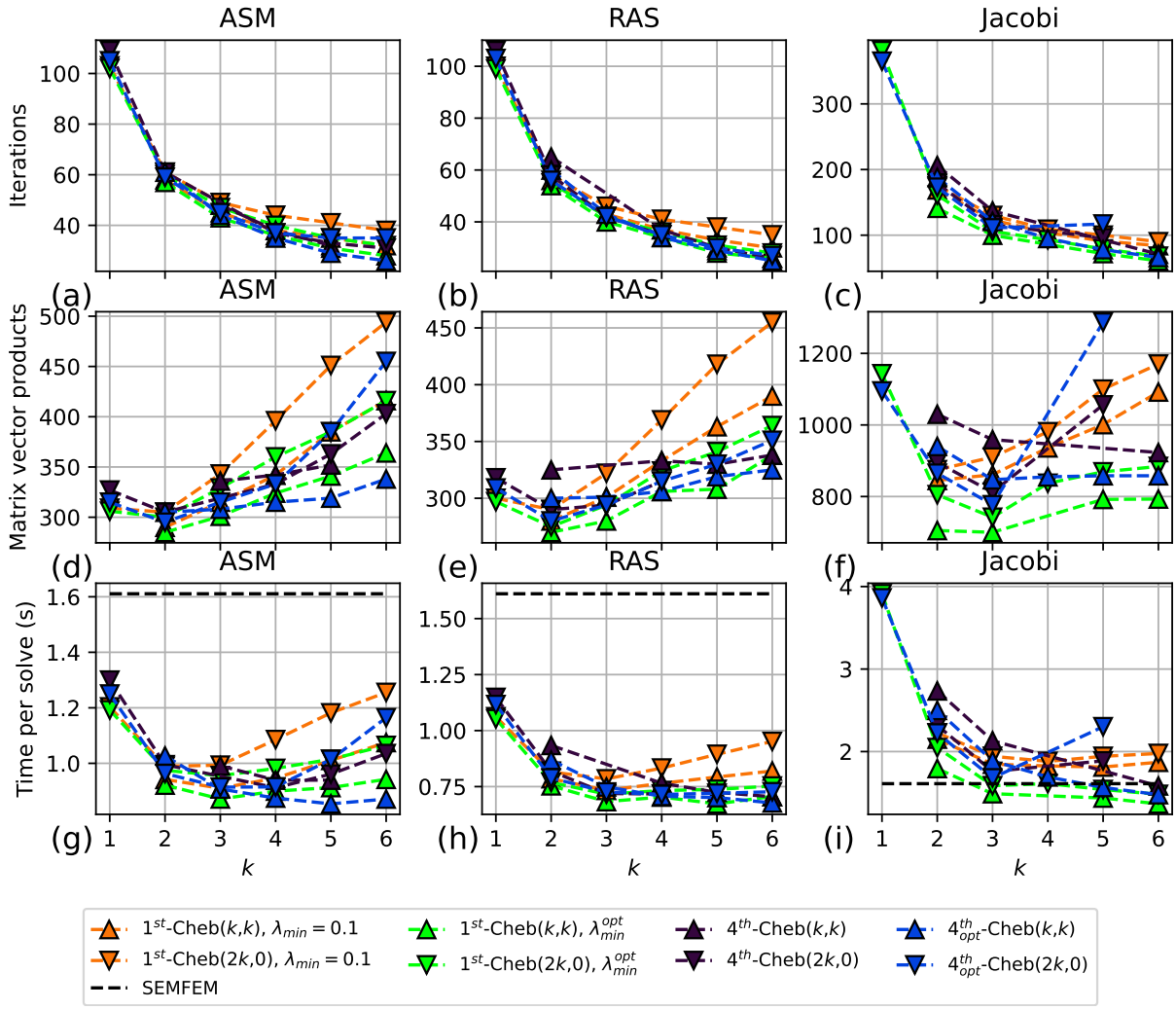



Figure 4.12: Kershaw results, $\varepsilon = 0.3$.

to Jacobi-based Chebyshev smoothing.

For $\varepsilon = 0.05$, the fastest time to solution is reached using SEMFEM preconditioning. The pMG methods are significantly more expensive. For the 4th and opt. 4th-kind Chebyshev-accelerated RAS schemes considered, the time to solution is lowered by increasing the order. Further, with the extreme geometric deformation, the multigrid approximation property constant eq. (4.2) is expected to be quite large for this case. The results in fig. 4.3 predict that the one-sided approach yields a better convergence rate. The results confirm this theoretical expectation.

A summary of the results for the Kershaw case is shown in table 4.7. This table reports the solver yielding the lowest time to solution (in seconds), $T_S$, the iteration count, and the speedup over the time to solution of the default `nekRS` solver, $T_D$. The ratio between the time spent doing coarse grid solves for the default solver, $(T_{crs})_D$, and the time spent doing

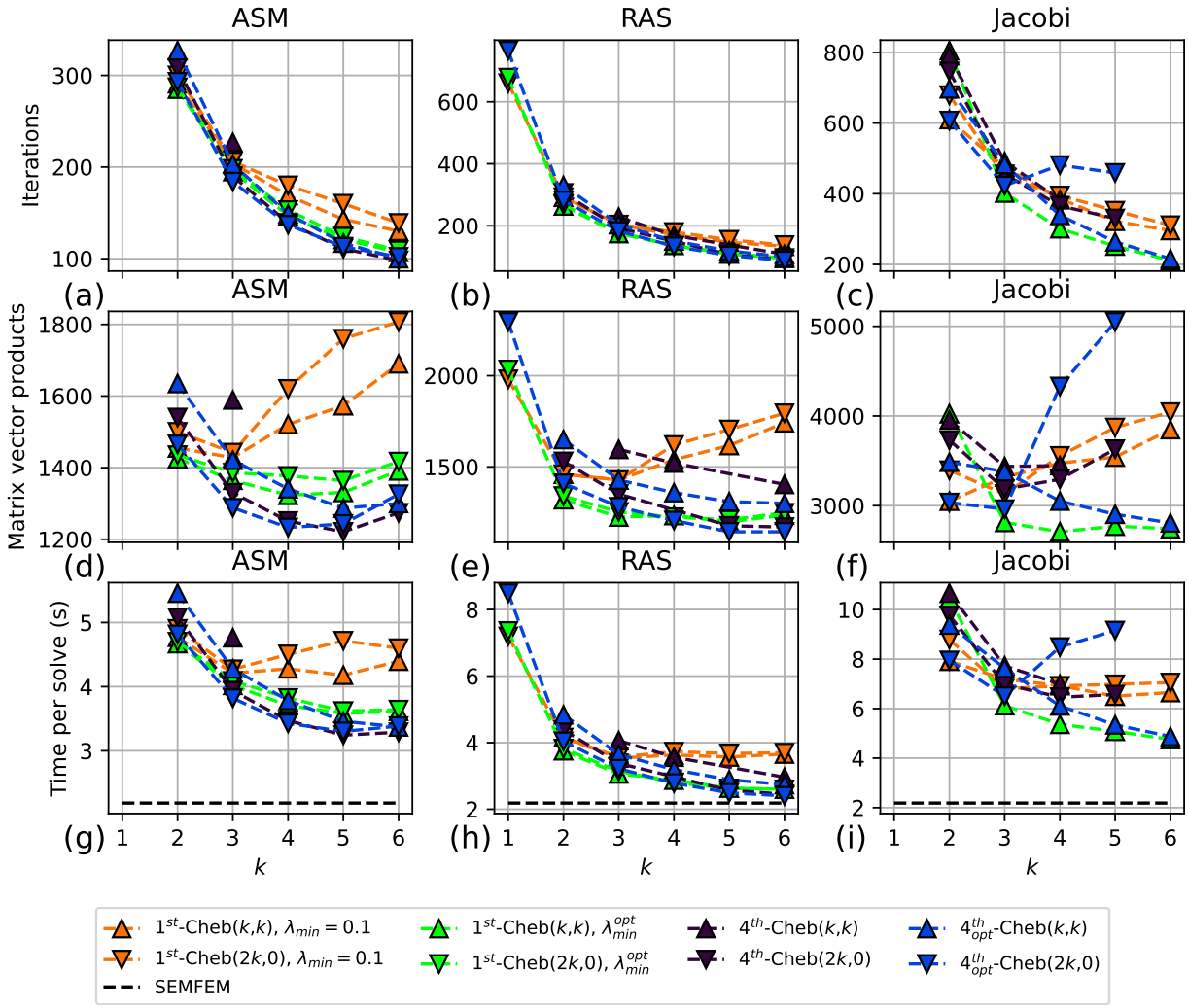



Figure 4.13: Kershaw results, $\varepsilon = 0.05$.

coarse grid solves for the fastest solver, $(T_{crs})_S$, is reported. The default `nekRS` solver is $1^{st}$-Cheb, ASM(3,3),(7,3,1). For the Kershaw case, these tables demonstrate that optimizing $\lambda_{min}$ in the 1st-kind Chebyshev scheme greatly improves the solver performance. While $\varepsilon = 1, 0.3$ do not benefit from the use of a one-sided V-cycle, the use of the one-sided V-cycle, in conjunction with the optimized 4th-kind Chebyshev smoother, is able to increase the solver speedup relative to the default solver by another 13% for $\varepsilon = 0.05$. A 75% speedup is achieved over the default solver for $\varepsilon = 1, 0.05$. For $\varepsilon = 0.3$, a much more modest 35% is achieved.

### 4.4.2 Navier-Stokes

Results for the 146, 1568, and 67 pebble cases are shown in fig. 4.14, fig. 4.15, and fig. 4.16, respectively. Based on the strong scalability study in section 2.9, the processor counts from table 2.6 are used. Since the 4th and optimized 4th-kind Chebyshev smoothers are comparable to the 1st-kind Chebyshev smoother with optimized $\lambda_{min}$ as shown in section 4.4.1, this smoother is omitted from these cases.

In the 146 pebble case, fig. 4.14 indicates that the optimal amount of Chebyshev smoothing is $k = 4$, which corresponds to 8th-order polynomial smoothing in the $(2k, 0)$ approach. This is observed, despite the small up-tick in the number of matrix-vector products required at that order, see fig. 4.14d at $k = 3$. In both the Jacobi-based and ASM-based Chebyshev smoothers, an overall reduction in the time per solve is observed in utilizing the $(2k, 0)$ V-cycle approach, especially for lower orders. However, around $k = 5$, the relative performance of the $(k, k)$ and $(2k, 0)$ V-cycle approaches switch. RAS-based Chebyshev smoothing, however, does not seem to benefit as much from the half V-cycle approach. pMG with a $4^{th}_{opt}$ Cheb RAS(4,4) smoother yields the fastest time to solution, as summarized in 4.7. Tuning the pMG parameters yields only a modest 17% speedup over the default preconditioning strategy in `nekRS`.

The lowest time to solution for the 1568 pebble case is achieved using $4^{th}$-Cheb, ASM(12,0), table 4.7. fig. 4.15d,g shows that the number of matrix-vector products remains nearly constant with respect to the order, yielding a lower time to solution by minimizing the coarse grid cost. The performance of the 1st-kind Chebyshev smoother, however, plateaus around $k = 3$, especially for the Schwarz-based smoothers, see fig. 4.15g,h. Although some improvement is observed through the use of the one-sided V-cycle, an additional benefit is observed by using the alternate Chebyshev smoothers from Lottes's work [23]. Without the added benefit of the one-sided V-cycle, the fastest solver for this case yields a 17% speedup over the default (table 4.6). However, enabling this one-sided V-cycle approach further increases the solver performance to a 27% speedup over the default solver (table 4.7).

The 67 pebble case is distinct from the other two pebble meshes. The 146 and 1568 pebble cases are meshed using an all-hex meshing strategy developed by Lan and coworkers [53]. The 67 pebble case includes chamfers, and is meshed using an alternate Voronoi cell approach [54]. The resulting mesh proves to be much more challenging for the pMG based strategies. As shown in table 2.3, mesh quality metrics, such as the min scaled Jacobian and max aspect ratio reveal the 67 pebble mesh to be much more difficult to solve than the 146 and 1568 pebble meshes. Nevertheless, pMG with one-sided Chebyshev smoothing prove promising for this case. The time to solution is minimized at relatively high orders

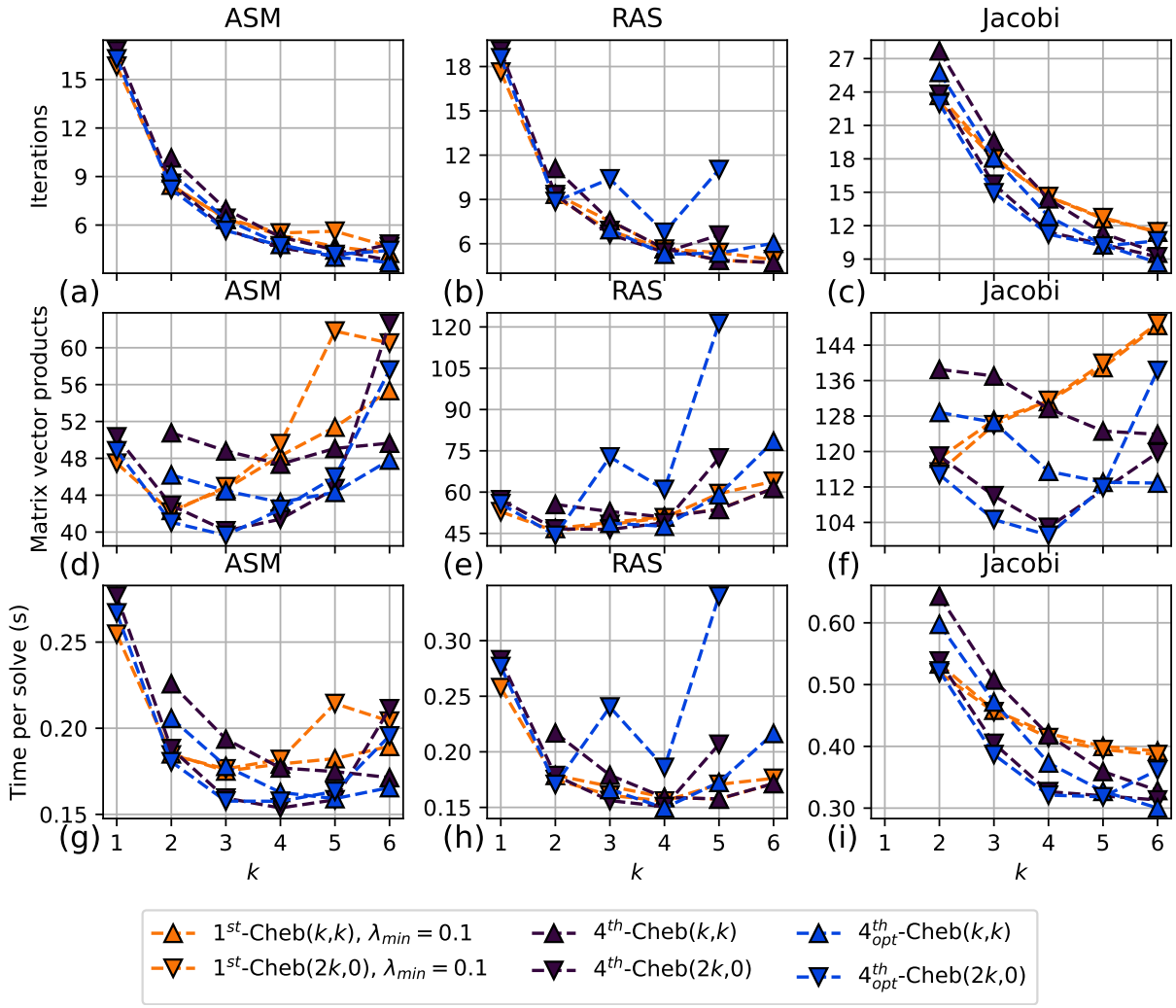



Figure 4.14: 146 pebble results.

with RAS-based Chebyshev. This is further improved through the use of the one-sided V-cycle. The *total work* required per solve is reduced at higher orders for the Schwarz-based approaches. Through tuning the pMG parameters, a significant speedup of 81% is achieved over the default solver. In this case, the fastest pMG preconditioner is $4^{th}_{opt}$-Cheb, RAS(12,0). However, the downward slope of the time to solution with respect to the order indicates that smoothing with an *even higher order* is likely advantageous in this case.

## 4.5 DIRECT OPTIMIZATION OF POLYNOMIAL SMOOTHERS

In section 4.1, two error bounds were employed to construct polynomial smoothers that are optimal with respect to the error bound. For example, the fourth-kind Chebyshev polyno-

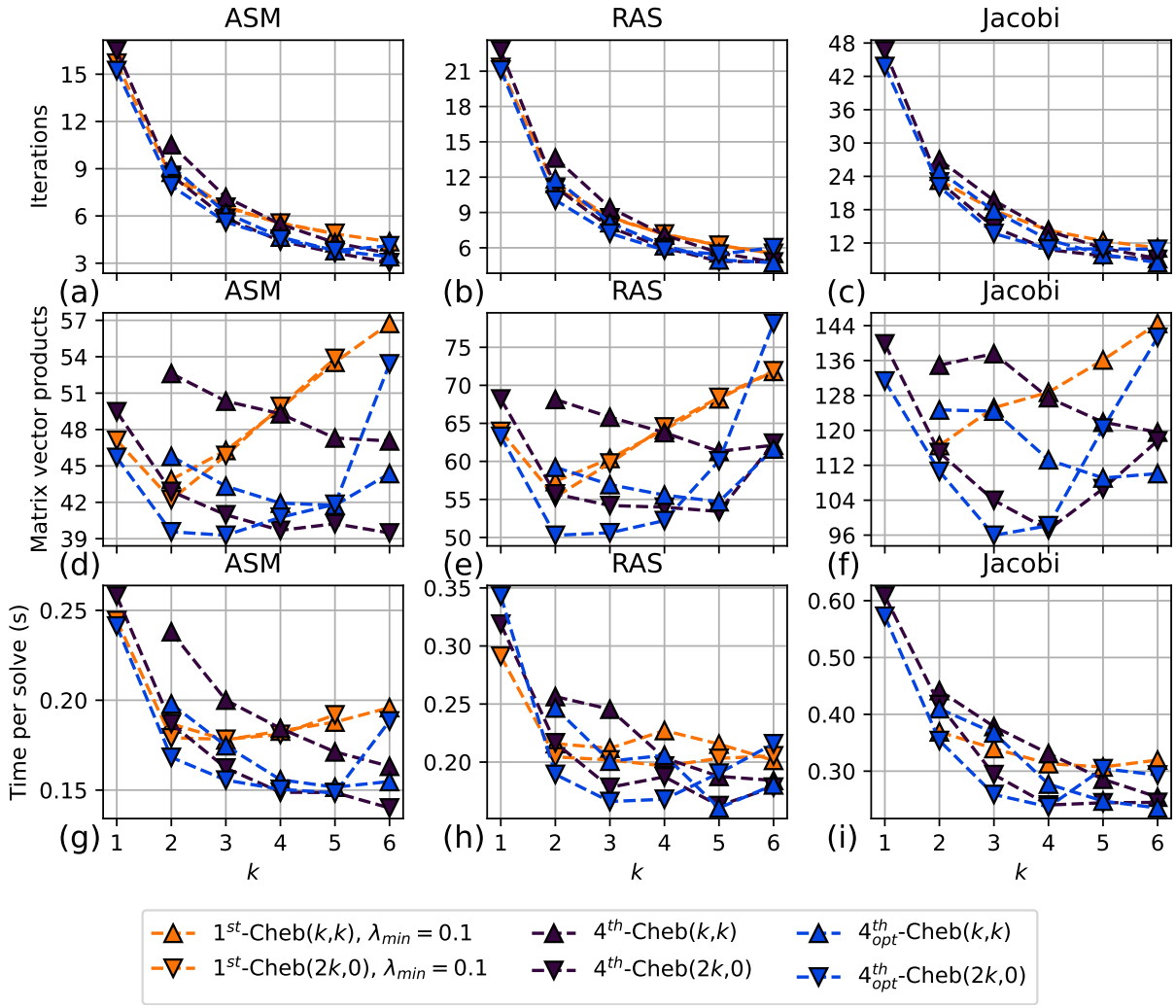



Figure 4.15: 1568 pebble results.

mials are the solution to the *weighted mini-max* problem posed in the *two-level* error bound from Hackbusch, eq. (4.1) [23]. Optimizing *multi-level* V-cycle error bound from lemma 4.1, a result due to Lottes [23], yields the optimized fourth-kind Chebyshev polynomial smoothers in eq. (4.8). Lemma 4.1 proves useful in analyzing the parameter space considered in section 4.1.2. For example, concrete solutions regarding *when* to apply the symmetric $(k, k)$ V-cycle versus the $(2k, 0)$ V-cycle are provided in the analysis in section 4.1.2, which is summarized in table 4.2. This analysis shows good agreement with the experimental results in section 4.2, fig. 4.3. The relative performance ranking of the various polynomial smoothers, moreover, is readily predicted by the bound in lemma 4.1, as demonstrated in figs. 4.7 and 4.8. However, as alluded to in figs. 4.7 and 4.8, the bounds in lemma 4.1 are not *sharp*. While the Local Fourier Analysis methods, presented in section 4.3 and also shown in figs. 4.7 and 4.8, provide a more accurate estimate of the multigrid error convergence

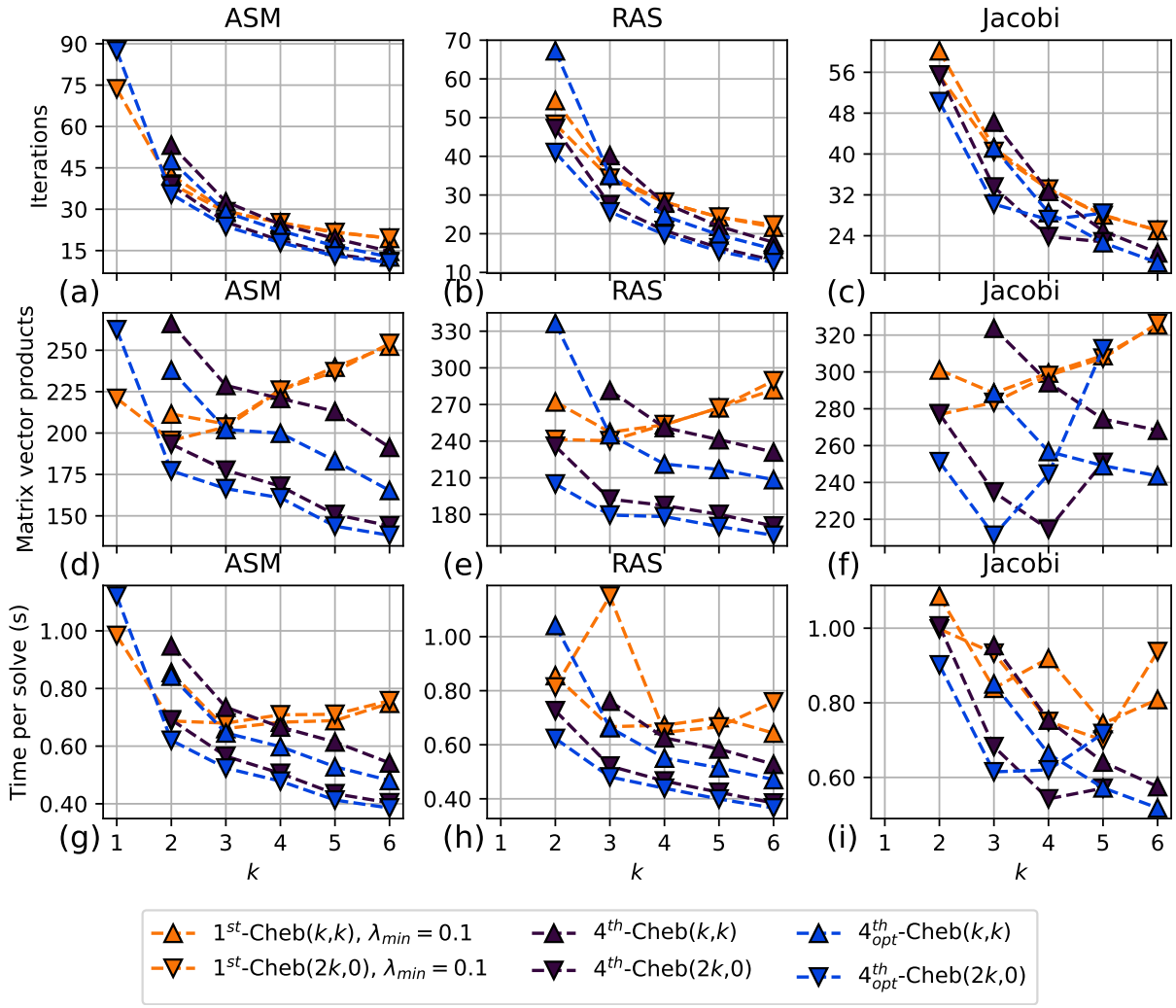



Figure 4.16: 67 pebble results.

factor, they nevertheless struggle to provide sufficiently *sharp* estimates in the presence of high aspect-ratio grids. The question, therefore, remains: *How can we construct optimal polynomial smoothers for a given case?*

In this section, we present a scheme that directly optimizes the polynomial smoother with respect to the observed solver convergence factor, rather than relying on the error bound in lemma 4.1. The optimization problem in question is to find the polynomial smoother the average convergence rate, eq. (4.24), for a preconditioned KSP solver. Each objective function evaluation, therefore, requires solving the system to the specified tolerance. Similar to the optimized fourth-kind Chebyshev polynomial smoother in eq. (4.8), the fourth-kind Chebyshev polynomials are a convenient basis for constructing an optimized polynomial smoother with respect to the average convergence rate. This has the added benefit of allowing us to leverage the same optimized fourth-kind Chebyshev polynomial smoother

Table 4.6: Solver configuration with the fastest time to solution, restricted to identical pre-/post-smoother orders. Processor counts identified from the strong scalability study in section 2.9 in table 2.6 are used.

| Case | Fastest Solver | $T_S$ | Iterations | $\frac{T_D}{T_S}$ | $\frac{(T_{crs})_D}{(T_{crs})_S}$ |
|---|---|---|---|---|---|
| Kershaw($\varepsilon = 1$) | $1^{st}$-Cheb, $\lambda_{min}^{opt}$, RAS(2,2) | 0.09 | 8 | 1.75 | 1.13 |
| Kershaw($\varepsilon = 0.3$) | $1^{st}$-Cheb, $\lambda_{min}^{opt}$, RAS(5,5) | 0.67 | 28 | 1.35 | 1.79 |
| Kershaw($\varepsilon = 0.05$) | $1^{st}$-Cheb, $\lambda_{min}^{opt}$, RAS(6,6) | 2.60 | 95 | 1.62 | 2.03 |
| 146 pebble | $4_{opt}^{th}$-Cheb, RAS(4,4) | 0.15 | 5.3 | 1.17 | 1.21 |
| 67 pebble | $4_{opt}^{th}$-Cheb, RAS(6,6) | 0.47 | 16.0 | 1.40 | 1.88 |
| 1568 pebble | $4_{opt}^{th}$-Cheb, ASM(5,5) | 0.15 | 3.8 | 1.17 | 1.69 |

Table 4.7: Solver configuration with the fastest time to solution. Processor counts identified from the strong scalability study in section 2.9 in table 2.6 are used.

| Case | Fastest Solver | $T_S$ | Iterations | $\frac{T_D}{T_S}$ | $\frac{(T_{crs})_D}{(T_{crs})_S}$ |
|---|---|---|---|---|---|
| Kershaw ($\varepsilon = 1$) | $1^{st}$-Cheb, $\lambda_{min}^{opt}$, RAS(2,2) | 0.09 | 8 | 1.75 | 1.13 |
| Kershaw ($\varepsilon = 0.3$) | $1^{st}$-Cheb, $\lambda_{min}^{opt}$, RAS(5,5) | 0.67 | 28 | 1.35 | 1.79 |
| Kershaw ($\varepsilon = 0.05$) | $4_{opt}^{th}$-Cheb, RAS(12,0) | 2.40 | 88 | 1.75 | 2.31 |
| 146 pebble | $4_{opt}^{th}$-Cheb, RAS(4,4) | 0.15 | 5.3 | 1.17 | 1.21 |
| 67 pebble | $4_{opt}^{th}$-Cheb, RAS(12,0) | 0.37 | 12.5 | 1.81 | 2.41 |
| 1568 pebble | $4^{th}$-Cheb, ASM(12,0) | 0.14 | 3 | 1.27 | 2.13 |

in eq. (4.8), albeit with different $\beta_i$ coefficients. Furthermore, alg. 4.1 can be used directly without any modifications to the several software implementations previously mentioned; only the $\beta_i$ coefficients need to be altered. Lastly, as the fourth-kind Chebyshev polynomial smoother is often either the best or competitive with the best polynomial smoother, a high-quality initial guess to the optimization problem is done by simply setting each $\beta_i = 1$. The polynomial smoothers produced via this method are denoted as "NM" in the results that follow.

While the description of the optimization problem is straightforward, how do we solve it during *run-time* for a particular case? The Nelder-Mead algorithm [89] is a simple, yet *robust*, algorithm for solving unconstrained optimization problems. The Nelder-Mead algorithm, alg. 4.3, is a direct search method that does not require the gradient of the objective function, which is not readily available in this case. A simple header-only C++ implementation of the Nelder-Mead algorithm, based on [90] and Burkhardt's implementation [91], is implemented

in the present work. An additional termination heuristic, which checks if the solver iteration count has not improved in the last $N$ function evaluations, is implemented to limit the Nelder-Mead algorithm from fruitlessly *over-optimizing* the polynomial smoother. $N = k + 5$ is chosen for the present work, where $k$ is the smoother polynomial order. This heuristic is motivated by the fact that the first $N$ function evaluations in the Nelder-Mead algorithm are required to construct the initial simplex prior to making any optimization progress. The Nelder-Mead algorithm is used to solve the optimization problem for the polynomial smoother coefficients in eq. (4.8), where the objective function is the average solver convergence rate, eq. (4.24).

### 4.5.1 Results

The Kershaw case with $\varepsilon = 0.3$ from fig. 2.10 in section 2.8.1 with $E = 36^3$, $p = 7$ is used to demonstrate the efficacy of directly optimizing the polynomial smoother. A single node of Summit with $P = 6$ V100 GPUs is used for the experiments as $n/P \sim 2.67M$ represents a case that is reasonably close to the strong-scaling limits for the solvers considered in section 2.9, specifically table 2.6. GMRES(15) is used as the KSP solver with multigrid as the preconditioner. The ASM smoother from section 2.6 is accelerated using the various Chebyshev polynomial smoothers, including the directly optimized polynomial, whose coefficients come from minimizing the average convergence factor through Nelder-Mead as described in this section. For tuning the coefficients of the directly optimized polynomial smoother, a random solution vector satisfying the boundary condition is used to generate the right-hand side, while the actual solve uses the right-hand side described in eq. (2.33). This ensures that the results from the optimization problem are not specific to the given right-hand side. The Nelder-Mead optimization algorithm is run for no more than 100 function evaluations, with the additional termination heuristic described in the preceding section providing an early termination condition.

Optimization progress in terms of the average convergence rate as a function of the number of function evaluations is shown in fig. 4.17. The initial convergence rate, $\rho_0$, which corresponds to the fourth-kind Chebyshev polynomial smoother, is used to normalize the objective function to show the relative improvement across several polynomial orders. Both the $(k, k)$ and $(2k, 0)$ multigrid schemes discussed in section 4.1.2 are considered. The improvement in the convergence rate quickly plateaus during the optimization process, requiring fewer than the prescribed 100 function evaluations. While the directly optimized polynomial smoother provides a modest improvement to the average convergence rate less than 10%, this improvement is compounded over each solver iteration. We note that, for a fixed polynomial order,

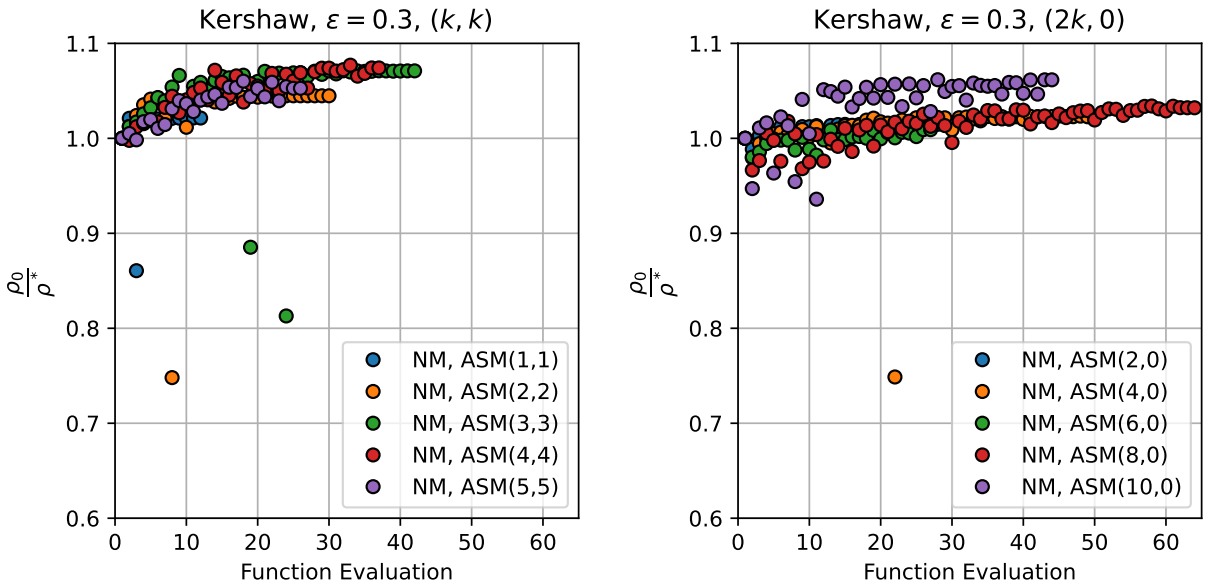



Figure 4.17: Nelder-Mead optimization progress as a function of the number of function evaluations for the Kershaw $\varepsilon = 0.3$ case. The average convergence factor of the initial guess, $\rho_0$ which corresponds to the fourth-kind Chebyshev polynomial smoother, is used to normalize the objective function. *Left* optimization progress for the $(k, k)$ smoother. *Right* optimization progress for the $(2k, 0)$ smoother.

the expense of applying a single iteration of alg. 4.1 does not depend on the coefficients $\beta_i$. The solver time speedup, therefore, is the same as the improvement in the iteration count, shown in fig. 4.18

While fig. 4.17 shows that the average convergence rate is only improved by a modest amount, the actual improvement in the iteration count is appreciable. For example, fig. 4.18 shows that the $(k, k)$ smoother with $k = 3$ reduces the iteration count by 20%, requiring relatively few function evaluations to achieve this improvement. We note that the function evaluations required in the optimization, while expensive compared to a single solve, are amortized over the many solves required in a production Navier-Stokes simulation, where $10^4 - 10^6$ solves are typical. This expense, further, is only incurred during the initial setup of the simulation. In a simulation campaign, a user can further reduce this expense by caching the optimized smoother coefficients from the initial setup and reusing them for subsequent simulations.

Figure 4.19 shows the result of the smoother polynomials optimized via Nelder-Mead for the Kershaw $\varepsilon = 0.3$ case, similar to fig. 4.1. An unintuitive result from fig. 4.19 that the optimized smoother polynomials may leave the highest frequency mode largely unsmoothed. The Chebyshev polynomials with $k \geq 2$ in fig. 4.1, for example, limit $p_k(\lambda = 1) \in [-1/3, 1/3]$.

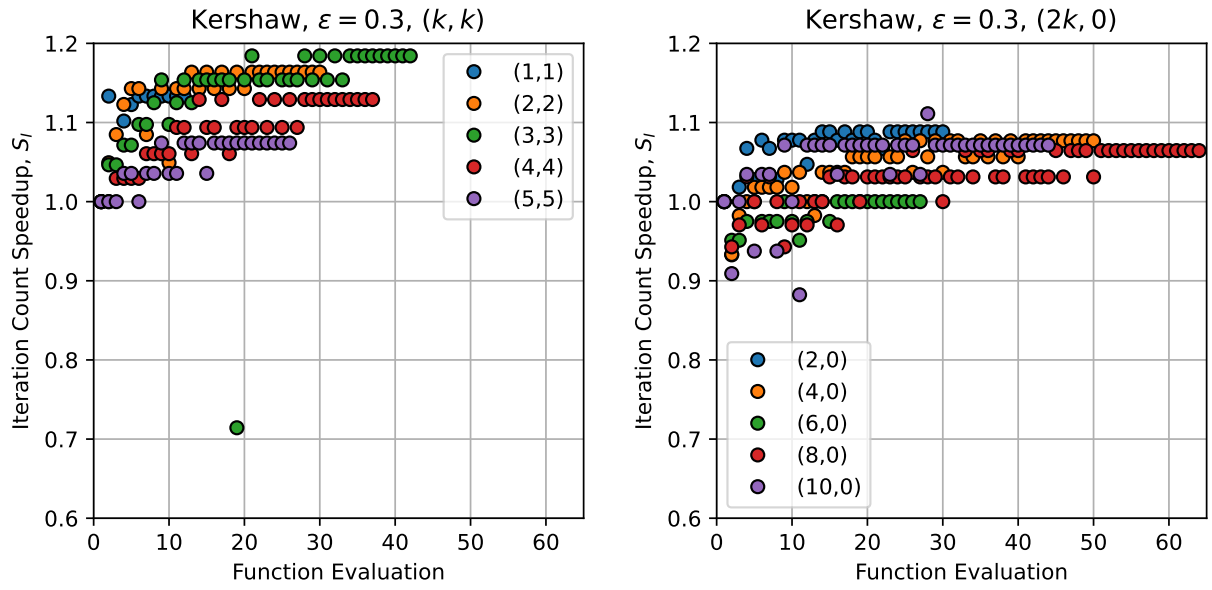



Figure 4.18: Improvement in iteration count during optimization. The iteration count for a given number of function evaluations is normalized by the iteration count of the fourth-kind Chebyshev polynomial smoother. *Left* iteration count improvement for the $(k, k)$ smoother. *Right* iteration count improvement for the $(2k, 0)$ smoother.

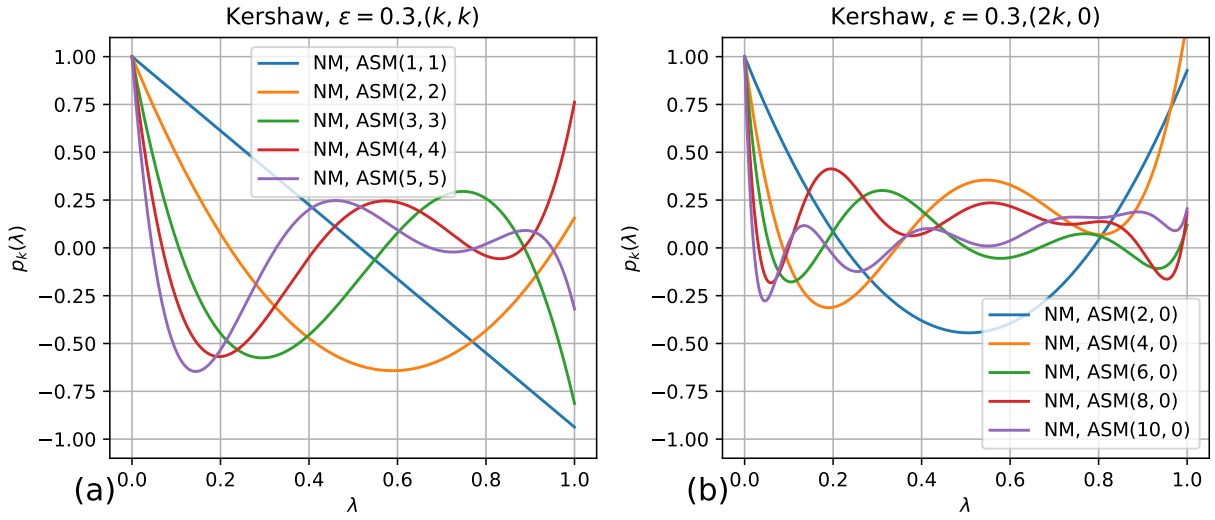



Figure 4.19: Directly optimized smoother polynomials on $\lambda \in [0, 1]$ from Kershaw $\varepsilon = 0.3$ case. Figure 4.19a optimized smoothers polynomials for $(k, k)$ multigrid scheme. Figure 4.19b optimized smoothers polynomials for $(2k, 0)$ multigrid scheme.

The optimized smoother polynomials, for example, tend to *discount* targeting the highest frequency mode to target intermediate and lower frequency modes. For example, the poly-

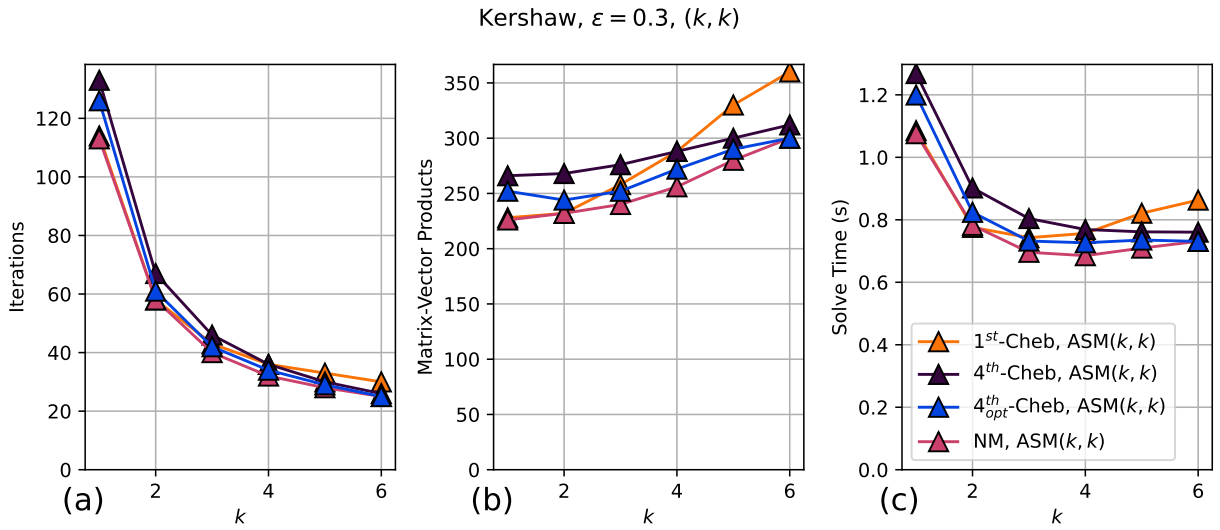



Figure 4.20: Iteration count, matrix-vector products, and solution time for the Kershaw $\varepsilon = 0.3$ case, using the $(k, k)$ multigrid scheme with various polynomial smoothers accelerating the ASM smoother.

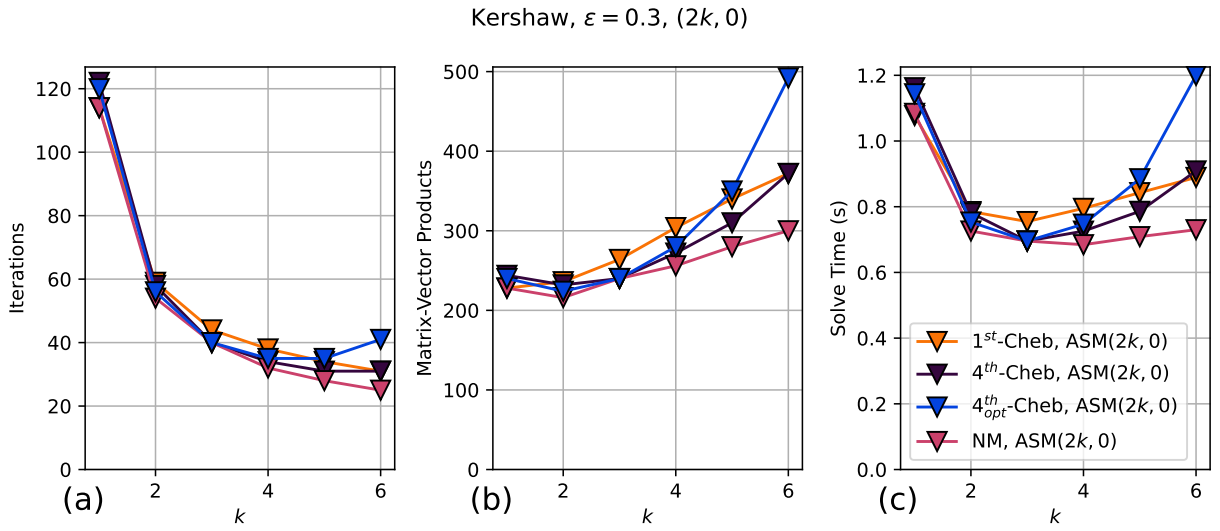



Figure 4.21: Iteration count, matrix-vector products, and solution time for the Kershaw $\varepsilon = 0.3$ case, using the $(2k, 0)$ multigrid scheme with various polynomial smoothers accelerating the ASM smoother.

nomial pertaining to the NM, ASM$(3, 3)$ smoother is less than -3/4 at $\lambda = 1$. One potential explanation for this phenomenon is that the Chebyshev polynomials displayed in fig. 4.1 are developed to target higher-frequency modes to reduce the maximum eigenvalue of the multigrid error propagator. The optimized smoother polynomials in fig. 4.1, on the other hand, directly optimize for the average convergence rate of a KSP solver utilizing multigrid

as a preconditioner. As such, the optimized smoother polynomials target as their objective function a proxy for the condition number. While the maximum eigenvalue of $M^{-1}A$ may remain large, the condition number can further be reduced by *increasing* the minimum eigenvalue.

To determine the overall improvement in the solver performance through the optimized polynomial smoothers as compared to the other Chebyshev polynomial smoothers, we consider the iteration count, matrix-vector products, and solution time for the Kershaw $\varepsilon = 0.3$ case in figs. 4.20 and 4.21. As noted, the right-hand side used in the optimization problem is random, while the actual solve results here are described in eq. (2.33). We observe that the optimized smoother polynomials outperform all other Chebyshev polynomial smoothers, irrespective of the polynomial order, for both the $(k, k)$ and $(2k, 0)$ multigrid schemes. While the optimized smoother polynomials offer only a modest improvement to the overall solver run-time, in certain cases, it improves the run-time by as much as 21% compared to the second-best polynomial smoother.

**Algorithm 4.3:** Nelder-Mead Simplex Method [88]

**Input:** Initial guess, $\underline{x}_0$; Step size $\underline{\Delta}x$ for initial simplex
**Output:** Approximate minimum, $\underline{x}^*$
**for** $i = 1$ **to** $n$ **do**
  $\underline{x}_i \leftarrow \underline{x}_0 + \underline{\Delta}x_i$ `/* construct initial simplex */`
**end**
**while** *not converged* **do**
  Sort points $\underline{x}_0, \ldots, \underline{x}_n$ such that $f(\underline{x}_0) \leq, \ldots, f(\underline{x}_n)$
  $\underline{m} \leftarrow \frac{1}{n}\sum_{i=0}^{n-1} \underline{x}_i$
  $\underline{r} \leftarrow 2\underline{m} - \underline{x}_n$ `/* generate` ***reflected*** `point */`
  **if** $f(\underline{x}_0) \leq f(\underline{r}) < f(\underline{x}_n)$ **then**
    $\underline{x}_n \leftarrow \underline{r}$
    **continue**
  **end**
  **if** $f(\underline{r}) \leq f(\underline{x}_0)$ **then**
    $\underline{s} \leftarrow \underline{m} + 2(\underline{m} - \underline{x}_n)$ `/* construct` ***expansion*** `point */`
    **if** $f(\underline{s}) < f(\underline{r})$ **then**
      $\underline{x}_n \leftarrow \underline{s}$
      **continue**
    **else**
      $\underline{x}_n \leftarrow \underline{r}$
      **continue**
    **end**
  **end**
  **if** $f(\underline{r}) \geq f(\underline{x}_{n-1})$ **then**
    **if** $f(\underline{r}) \leq f(\underline{x}_n)$ **then**
      $\underline{c}_O \leftarrow \underline{m} + \frac{1}{2}(\underline{r} - \underline{m})$ `/* construct` ***outside contraction*** `point */`
      **if** $f(\underline{c}_O) < f(\underline{r})$ **then**
        $\underline{x}_n \leftarrow \underline{c}_O$
        **continue**
      **end**
    **else if** $f(\underline{r}) \geq f(\underline{x}_n)$ **then**
      $\underline{c}_I \leftarrow \underline{m} + \frac{1}{2}(\underline{x}_n - \underline{m})$ `/* construct` ***inside contraction*** `point */`
      **if** $f(\underline{c}_I) < f(\underline{x}_n)$ **then**
        $\underline{x}_n \leftarrow \underline{c}_I$
        **continue**
      **end**
  **end**
  **for** $i = 1$ **to** $n$ **do**
    $\underline{x}_i \leftarrow \underline{x}_0 + \frac{1}{2}(\underline{x}_i - \underline{x}_0)$ `/* shrink simplex */`
  **end**
**end**

## Chapter 5: Hiding Coarse-Grid Solves through Overlapping

In chapter 3, weak scaling studies highlight the inherent $\log P$ communication complexity associated with the coarse-grid solve. This limits the parallel scalability of the geometric $p$-multigrid-based preconditioner strategies from section 2.6. As we have observed in section 2.9 and chapter 4, however, one technique for mitigating the cost of coarse-grid solves is to limit the number of coarse-grid solves required. In chapter 4, we are able to achieve this by increasing the smoother polynomial order and exploiting polynomial smoothers which have provable upper bounds on the maximum eigenvalue of the multigrid error propagator which scale as $O(k^{-2})$ with respect to the polynomial order, $k$. Through effectively controlling the coarse-grid solve cost, we are able to develop more performant preconditioners for the SEM-discretized Poisson equation.

In this chapter, we propose a different technique to mitigate the relative expense of the coarse-grid solves by overlapping the coarse-grid solves. This is achieved by treating the coarse-grid problem as an *additive* correction to the geometric $p$-multigrid preconditioner, which otherwise remains multiplicative. The coarse-grid solve, which occurs on the CPU, occurs *in parallel* with the remaining portion of the multigrid V-cycle, which executes on the GPU. This allows us to effectively hide the cost of the coarse-grid solve behind the cost of the remaining portion of the V-cycle.

Additive multigrid schemes, at the same time, tend to degrade the convergence rate of the preconditioner in comparison to multiplicative schemes. This is further discussed in the theoretical considerations of section 5.1. In addition, section 5.2 presents a technique to help recover part of the convergence rate degradation by balancing the relative cost of the coarse-grid solve with the remaining portion of the V-cycle. We further note that the particular additive multigrid variant proposed herein is especially well-suited for heterogeneous architectures in that the *additive* portion of the preconditioner is *minimal*.

The organization of this chapter is as follows. The motivation and some theoretical considerations for this method are presented in section 5.1. In section 5.2, algorithmic details for the method are presented. The algorithm proposed in this chapter is Finally, in section 5.3, we present numerical results for the multigrid method with overlapped coarse-grid solves on small, model problems. To enable comparison with the other methods discussed in this thesis, large-scale performance studies for the cases described in section 2.8 are deferred to chapter 7.

## 5.1 MOTIVATION AND THEORY

During the execution of the geometric $p$-multigrid preconditioner described in alg. 2.8 from section 2.6, the coarse-grid solve is performed after all former smoother, residual re-evaluations, and restriction operations have been completed. At the typical scaling limit of $n/P \sim 2.5M$ degrees of freedom per GPU, established in section 2.9, the number of degrees of freedom corresponding to the $p = 1$ coarse grid problem scales with the number of elements per rank, $E/P$. For example, for a discretization at $p = 7$, the coarse grid problem would contain only ∼8,000 degrees of freedom per rank. This is *too little work* to justify running an AMG solver on the GPU, so the CPU is used instead. However, during the coarse-grid solve, the GPU is idle. This problem is further exacerbated by the fact that the coarse-grid solve requires non-trivial communication patterns which generally scale as $\log(P)$ [74], see chapter 3 for more details. All other operations in the V-cycle, however, utilize highly-scalable nearest neighbor exchanges. Thus, the coarse-grid solve is a bottleneck in the overall V-cycle performance, *especially at scale*.

How can we mitigate this problem? We propose applying the coarse-grid correction *additively* as shown in fig. 5.1. The use of additive V-cycles to offer better parallelism has been well studied [21, 22, 92, 93]. A major concern with the use of additive V-cycle methods is the degradation in the solver convergence rate in comparison to the multiplicative counterpart. In an additive V-cycle approach, the coarse component is added to the current level in the cycle. This has the effect of "over-correcting" the coarse-grid correction. This issue can compromise the ability to use the additive multigrid technique as a *solver*—the multigrid error propagator is no longer a contractor. This, however, does not prevent the additive V-cycle from being employed as a high-quality *preconditioner*. In the oldest additive multigrid scheme from Greenbaum [92], Greenbaum addresses this issue by applying each grid correction sequentially, such that each correction is made orthogonal to the residual after correction. This approach is considered in a subsequent paragraph in this thesis in eq. (5.8).

In contrast to adding a "correction factor" to the coarse-grid correction considered by Greenbaum, Bramble, Pasciak, and Xu, ignore this issue, instead choosing to use the multigrid method as a preconditioner [21]. Here, we summarize the preconditioner proposed by Bramble, Pasciak, and Xu using the notation from section 2.5. The seminal Bramble-Pasciak-Xu (BPX) algorithm [21] forms, for each level $j = 0, \ldots, \ell - 1$, the inverse of the symmetrized smoother

$$I - \Lambda_j^{-1} A_j = A_j^{-1} G_j^T A_j G_j, \tag{5.1}$$

where $G_j$ is the smoother iteration matrix and $A_j$ is the operator on level $j$. The BPX multi-level preconditioner is given by

$$M_{BPX}^{-1}\underline{r} = P_\ell A_\ell^{-1} P_\ell^T + \sum_{j=0}^{\ell-1} P_j \Lambda_j P_j^T \underline{r}, \tag{5.2}$$

where each $P_j$ is the prolongation operator from level $j$ to the finest level. While the BPX preconditioner built on geometric multigrid for Poisson is spectrally equivalent to the Poisson operator, it converges much slower than the corresponding multiplicative V-cycle [22].

Further improvements to the BPX algorithm, however, have been considered. For example, Vassilevski and Yang [22] suggest an improvement utilizing the "smoothed" two-level interpolation operators

$$\bar{P}_{j+1}^j = G_j P_{j+1}^j, \tag{5.3}$$

where $P_{j+1}^j$ prolongates from level $j+1$ to $j$. By composing the interpolants for all $i > j$ and letting $\bar{P}_j^j = I$,

$$\bar{P}_i^j = \bar{P}_{j+1}^j \bar{P}_{j+2}^{j+1} \dots \bar{P}_i^{i-1}, \tag{5.4}$$

a mathematically equivalent way to express the standard multiplicative V-cycle preconditioner in an additive fashion is

$$M_V^{-1}\underline{r} = \sum_{j=0}^{\ell} \bar{P}_j^0 \Lambda_j \bar{P}_j^0 \underline{r}. \tag{5.5}$$

This method is referred to as *multadd*. Vassilevski and Yang also propose a *simplified multadd* method, where $\Lambda_j$ in eq. (5.5) is replaced by the smoother $G_j$. The *multadd*, *simplified multadd*, and BPX methods are not, however, applicable in the matrix-free $p$-multigrid context, as they rely on being able to explicitly form the $\Lambda_k$ and/or $\bar{P}_{j+1}^j$ operators.

Another major difference in the scheme proposed herein is the use of an additive coarse-grid correction while treating the remainder of the V-cycle as multiplicative. The motivation behind this choice is that the coarse-grid solve the remainder of the multigrid V-cycle operate on different *execution spaces*, with the former being performed on the CPU, while the latter is done on the GPU. This allows overlapping the coarse-grid solve with the remainder of the V-cycle while *minimizing* the convergence rate degradation as the fewest possible levels are treated additively.

The proposed algorithm is described in more detail in section 5.2. Mathematically, the

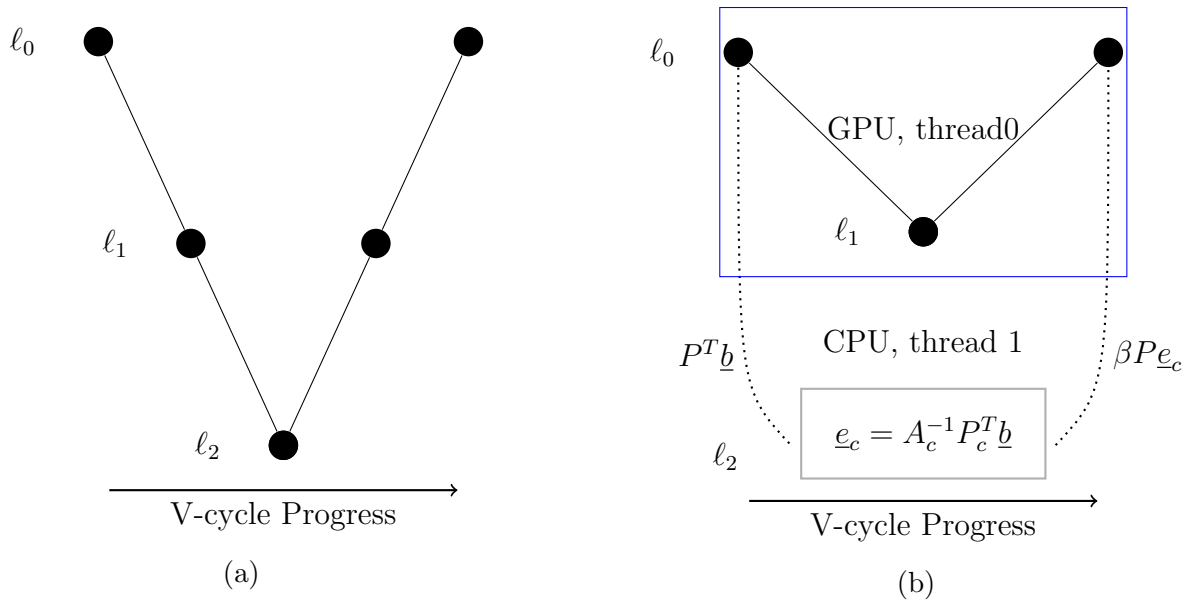



Figure 5.1: Standard, multiplicative multigrid (fig. 5.1a) versus multiplicative multigrid with additive coarse-grid correction (fig. 5.1b). The coarse-grid solve, shown in the dashed box in fig. 5.1b, is performed on the CPU simultaneous with the rest of the multigrid V-cycle on the GPU, shown in the solid box. The parameter $\beta$ is introduced to prevent over-correcting the solution.

proposed method depicted in fig. 5.1b and alg. 5.1 corresponds to

$$M_{AC}^{-1}\underline{r} = \beta P_\ell A_\ell^{-1} P_\ell^T \underline{r} + M_0^{-1}\underline{r}, \tag{5.6}$$

where $P_\ell$ is the prolongation operator from the coarsest level to the finest level. $M_0^{-1}$ is recursively defined as in eq. (2.17) for levels $j = 0, \ldots, \ell - 2$,

$$I - M_j^{-1}A_j = G_j^T \left( I - P_{j+1}^j M_{j+1}^{-1} \left(P_{j+1}^j\right)^T A_j \right) G_j, \tag{5.7}$$

where each $G_j$ is the smoother iteration on level $j$, The recursion terminates with $M_{\ell-1}^{-1} = G_{\ell-1}$, which corresponds to only smoothing on level $\ell-1$. A coarse-grid correction parameter, $\beta$, is introduced. In Greenbaum [92], $\beta$ is defined such that the coarse-grid correction is orthogonal to the residual after correction. Denoting $\underline{e}_\ell = A_\ell^{-1}P_\ell^T\underline{r}$, the correction due to Greenbaum, $\beta_G$, is given by

$$\beta_G = \frac{(\underline{r} - AM_0^{-1}\underline{r})^T(P_\ell\underline{e}_\ell)}{(P_\ell\underline{e}_\ell)^T A(P_\ell\underline{e}_\ell)}. \tag{5.8}$$

If $\beta_G$ is near unity, then the coarse-grid correction is already nearly orthogonal to the residual

after correction. Computation of $\beta_G$ requires two additional matrix-vector products and two dot products which can be performed within a single two-word all-reduce. While the computation of $\beta_G$ requires additional work *per cycle*, including this correction improves the convergence rate of the additive method. This trade-off is further explored in the results in section 5.3.

We note that the proposed method in eq. (5.6) is especially attractive for heterogeneous architectures, as the coarse-grid solve, which occurs on the CPU, can be overlapped with the remainder of the V-cycle, which occurs on the GPU. While the BPX method in eq. (5.2) shares this trait, the loss in convergence rate is minimized by our proposed method in eq. (5.6) by only treating a single level additively. The change from the multiplicative V-cycle to the additive coarse-grid correction proposed here is illustrated in fig. 5.1.

What effect on the convergence rate does the additive multigrid approach described in eq. (5.6) have? Notay and Napov [94] demonstrated that, provided the correct $\beta$ coefficient associated with the coarse-grid correction, the following condition number relating a multiplicative two-level preconditioner, $M^{-1}$, and its additive two-level counterpart, $M_{AC}^{-1}$,

$$\kappa(M^{-1}A) \leq \kappa(M_{AC}^{-1}A) \leq 4\kappa(M^{-1}A). \tag{5.9}$$

For large $\kappa(M^{-1}A)$, eq. (5.9) implies that the iteration count for the additive method is at most twice that of the multiplicative method. Equation (5.9), further, is not applicable for the multilevel preconditioner considered here. Provided that the coarse-grid solve and the remainder of the V-cycle are perfectly overlapped, the maximum speedup achievable using the approach proposed in eq. (5.6) is then a factor of two. What about when one component of the V-cycle is more expensive than the other? In this case, additional work can be shifted to either the coarse-grid solve or remainder of the V-cycle *at no additional run-time cost*, as the time to execute a single V-cycle of the method in eq. (5.6) is given as the maximum of the time to execute the coarse-grid solve and the time to execute the remainder of the V-cycle. This idea is further expanded in section 5.2.

## 5.2 ALGORITHM

The multiplicative V-cycle with additive coarse-grid correction in eq. (5.6) is described in alg. 5.1. The first step of the algorithm is to restrict the right-hand side to the coarsest level of the V-cycle. Once available, a thread parallel-block with two threads (or some partition of many threads) is created. One group of threads (e.g. thread 1) is responsible for executing the coarse-grid solve on the CPU, while the other group (e.g., thread 0) is

responsible for executing the remainder of the V-cycle. Once both operations are complete (thereby requiring a synchronization on the GPU to ensure all kernels have finished), the coarse-grid correction is added to the finest-level solution. In alg. 5.1, a correction factor $\beta$ for the coarse-grid correction is required. In this thesis, we consider $\beta = 1$ and $\beta = \beta_G$ as prescribed by Greenbaum [92] in eq. (5.8). The actions of alg. 5.1 are depicted on the right in fig. 5.1.

Our implementation of alg. 5.1 requires both OpenMP [95] and a fully-thread safe MPI implementation (`MPI_THREAD_MULTIPLE`). The former requirement comes from needing at least one thread per MPI rank to execute the coarse-grid solve and at least one thread per MPI rank for dispatching the kernel launches and communication operations required in the remainder of the V-cycle. The latter requirement, moreover, comes from the need to execute MPI commands *both* from the coarse-grid solve context *and* from the remainder of the V-cycle. While the development of a fully-thread safe MPI implementation is not without issues [96], more recent works (e.g. [97, 98]) have shown that the performance impact of supporting a fully thread-safe MPI implementation is not too onerous. In our use-case, we have not found any significant performance impact in the common solver operators considered in fig. 3.19.

What expected performance improvements can be achieved by the proposed method? For the purposes of analysis, let us consider the case with $\beta = 1$. The time to solution is written as

$$\begin{aligned} T_s &= n_{iter} T_{MG} \\ &= n_{iter}(T_{crs} + T_v), \end{aligned} \tag{5.10}$$

where $T_{MG}$ is the time to perform a single multigrid V-cycle, which is further decomposed into the coarse grid cost per iteration, $T_{crs}$, and the remainder of the V-cycle, $T_V$. Since no additional work in terms of matrix-vector products or smoother applications is introduced by the additive coarse grid method in eq. (5.6), the time to solution for overlapping the coarse grid solve with the remainder of the V-cycle is given by

$$T_s' = n_{iter}' \max(T_{crs}, T_v). \tag{5.11}$$

The speedup is therefore

$$S = \frac{n_{iter}(T_{crs} + T_v)}{n_{iter}' \max(T_{crs}, T_v)}. \tag{5.12}$$

**Algorithm 5.1:** Multiplicative Multigrid with Additive Coarse-Grid

**Data:** Right-hand side, $\underline{b}$
**Result:** Preconditioned Solution, $\underline{z}$
$\underline{b}_\ell \leftarrow P_\ell^T \underline{b}$ `/* Restrict right-hand side coarsest-level */`
**Thread Block**
  **Thread 1**
    $\underline{e}_\ell \leftarrow A_\ell^{-1} \underline{b}_\ell$ `/* Non-blocking solve */`
  **Thread 0**
    `/* Unwind recursion to get down-leg of V-cycle */`
    **for** $l = 0, \ldots, \ell - 2$ **do**
      $\underline{x}_l \leftarrow$ `pre-smooth`$(\underline{x}_l, \underline{b}_l)$ `/* e.g., algs. 2.12 and 4.1 */`
      $\underline{r}_l \leftarrow \underline{b}_l - A_l \underline{x}_l$ `/* Residual re-evaluation */`
      $\underline{b}_{l+1} \leftarrow \left(P_{l+1}^l\right)^T \underline{r}_l$ `/* Coarsen */`
    **end**
    $\underline{x}_{\ell-1} \leftarrow$ `pre-smooth`$(\underline{x}_{\ell-1}, \underline{b}_{\ell-1})$ `/* Smooth on second coarsest level */`
    `/* Unwind recursion to get up-leg of V-cycle */`
    **for** $l = \ell - 2, \ldots, 0$ **do**
      $\underline{x}_l \leftarrow \underline{x}_l + P_{l+1}^l \underline{x}_{l+1}$ `/* Add coarse-grid correction */`
      $\underline{x}_l \leftarrow$ `post-smooth`$(\underline{x}_l, \underline{b}_l)$
    **end**
`/* Threads synchronized outside of block */`
$\underline{x}_0 \leftarrow \underline{x}_0 + \beta P_\ell \underline{e}_\ell$ `/* Additive coarse-grid correction on finest-level */`
$\underline{z} \leftarrow \underline{x}_0$
**return** $\underline{z}$

If $n'_{iter} = n_{iter}$, then the maximum possible speedup by treating the coarse grid solve additively is a factor of two, as shown in fig. 5.2a. This maximum speedup, however, is not achieved due to the convergence degradation of the additive scheme compared to the multiplicative. An important observation, however, comes in the case when $T_{crs} > T_v$. This is representative of large cases, such as the $n = 51B, P = 27,648$ V100 GPU case presented in table 1.1 [16]. In this case, the cost of the remainder of the V-cycle can be increased through using additional levels in the multigrid hierarchy or increasing the number of smoother iterations without increasing the time to apply a single V-cycle. Let us denote this alternate V-cycle cost as $T'_v$. As shown in fig. 5.2b, $T'_v$ can be increased to $T'_v = T_{crs}$ without incurring any additional cost. This has the added benefit of reducing the number of iterations required to converge. In this scenario, if the additive coarse-grid variant yields $n'_{iter} < n_{iter}$ iterations

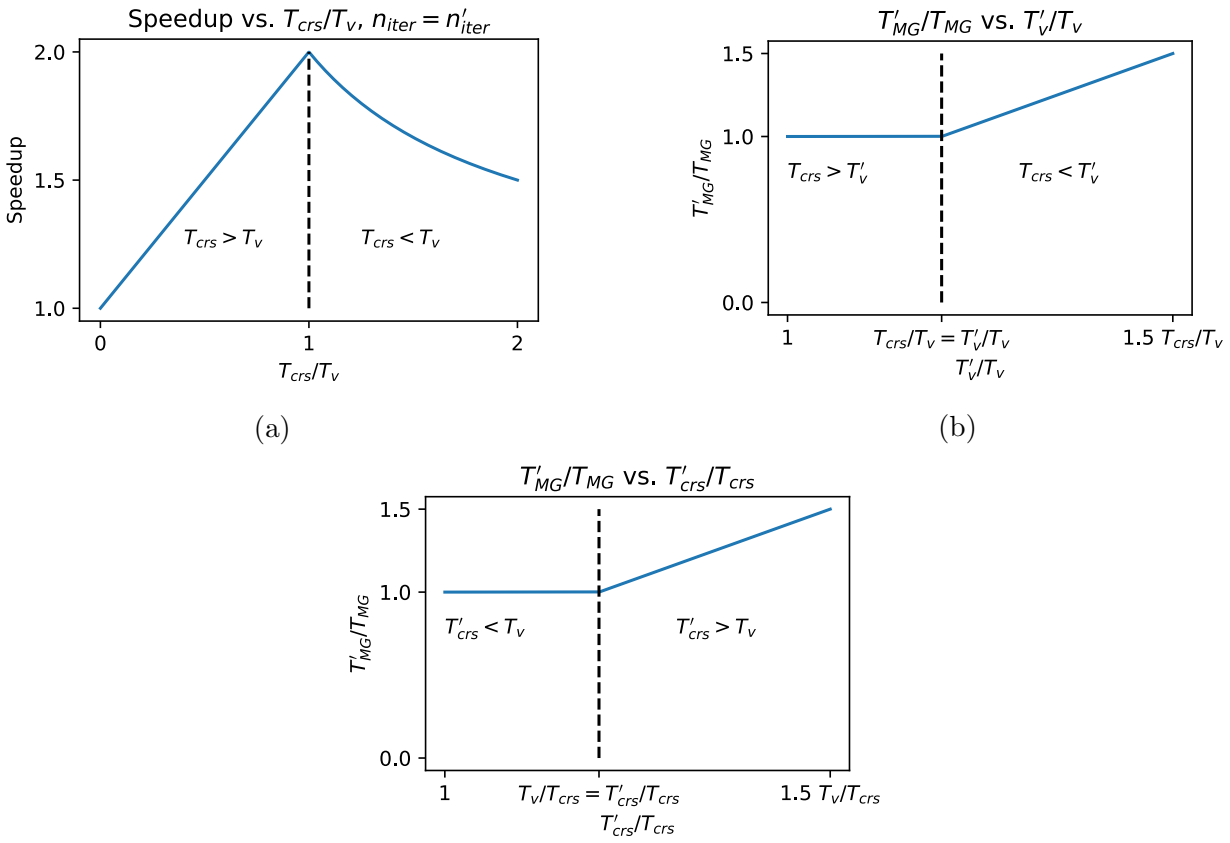



Figure 5.2: Figure 5.2a: speedup of additive coarse grid solve as a function of the ratio of the coarse grid cost per iteration to the remainder of the multigrid V-cycle per iteration, assuming $n_{iter} = n'_{iter}$. Figure 5.2b: cost per V-cycle evaluation for additive coarse-grid solve as a function of increasing smoother cost, assuming $T_v < T_{crs}$. Figure 5.2c: cost per V-cycle evaluation for additive coarse-grid solve as a function of increasing coarse-grid cost, assuming $T_{crs} < T_v$.

at $T'_v = T_{crs}$ and $T_{crs} > T_v$, then the speedup over the multiplicative V-cycle is

$$\begin{aligned} S &= \frac{n_{iter}}{n'_{iter}} \frac{T_v + T_{crs}}{T_{crs}} \\ &> \frac{n_{iter}}{n'_{iter}}. \end{aligned} \tag{5.13}$$

Let us also consider the case for sufficiently small problems and heavy smoothers/V-cycles, such that $T_v > T_{crs}$. In this case, the coarse-grid cost can be increased to the point $T_v = T'_{crs} > T_{crs}$. Consider, for example, that only a single AMG V-cycle is used to approximate the coarse-grid solution, as discussed in section 2.6. However, the accuracy of the coarse-grid correction can be improved at no additional cost per iteration, provided $T'_{crs} \leq T_v$, as

shown in fig. 5.2c. This method has potential not only to improve solver performance in the case where the coarse-grid cost grows, but at the other end of the spectrum, where the coarse-grid cost is small compared to the remainder of the V-cycle. Tuning for the selection of the V-cycle or coarse-grid solver during run-time, in the spirit of the tuner proposed in section 3.1, is planned for future work.

Numerical results for the half-cylinder case from section 2.4.1 are presented in section 5.3. The results are included to demonstrate the effect of the additive method on the convergence rate of the solver, as well as whether or not the correction factor for the coarse-grid correction as proposed by Greenbaum [92] in eq. (5.8) is required. Identifying the scenarios in which overlapping the coarse-grid solve with the remainder of the multigrid V-cycle is deferred to chapter 7.

## 5.3 NUMERICAL RESULTS

Numerical results demonstrating the efficacy of the proposed additive multigrid method in alg. 5.1 are presented in this section. Here, the half-cylinder cases briefly considered in section 2.4.1 are reconsidered here. In each case, the initial residual norm is reduced by a factor of $10^8$. The number of iterations and *fine level* matrix-vector products required to achieve this reduction are reported. Polynomial smoothers from section 2.7 and section 4.1 are employed to accelerate the ASM smoother alg. 2.10. To facilitate comparison with other methods, the larger-scale cases considered in section 2.8 are deferred to chapter 7.

The results in this section are divided into two parts. First, the effect of the choice of $\beta$ from alg. 5.1 are considered in section 5.3.1. In this thesis, we consider $\beta = 1$ and the Greenbaum correction $\beta = \beta_G$ from eq. (5.8). While the latter tends to improve the convergence of the solver, computing the correction requires two additional matrix-vector products *per V-cycle iteration*. Second, the performance of the proposed method is compared against the multiplicative V-cycle in section 5.3.2.

### 5.3.1 Effect of Correction-Factor for Coarse-Grid Corrections

Iteration count and *fine-level* matrix-vector products required to reach convergence using the additive V-cycle method are shown in figs. 5.3 to 5.6. Two values for $\beta$, the correction factor for the coarse-grid solve, are compared: $\beta = 1$ and $\beta = \beta_G$ eq. (5.8). The latter requires an additional two matrix-vector products *per V-cycle*.

The first result we note is that applying the Greenbaum correction $\beta = \beta_G$ either improves the convergence rate or has no effect compared to $\beta = 1$. For example, fig. 5.3 shows that,

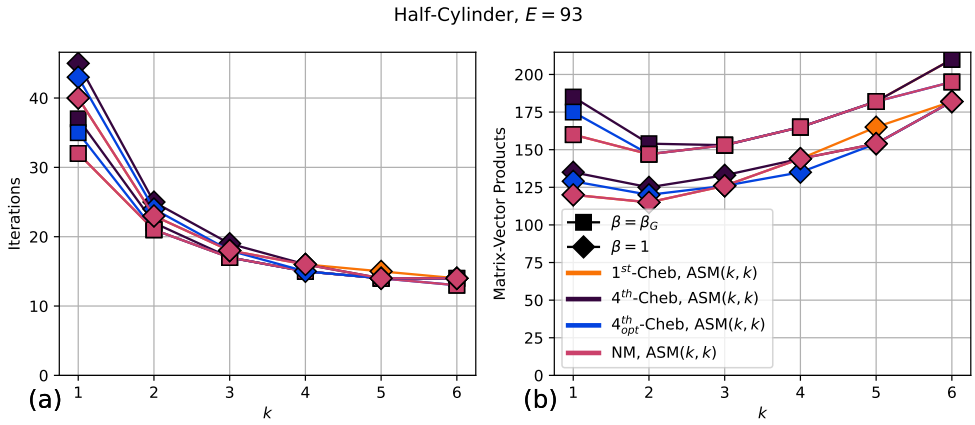



Figure 5.3: Convergence and matrix-vector product count comparison for the additive V-cycle with $\beta = 1$ and $\beta = \beta_G$, $E = 93$.

at $k = 1$, using $\beta = \beta_G$ *reduces* the iteration count by roughly 10 iterations. However, the improvement in the iteration count diminishes as the smoother polynomial order increases. Further, the reduction in the iteration count provided by using $\beta = \beta_G$ is not sufficient to make up for the additional work required to compute the correction. In the results considered later in chapter 7, however, reducing the iteration count may be sufficient to justify the additional work required. As such, using $\beta = \beta_G$ is not ruled out as a viable option.

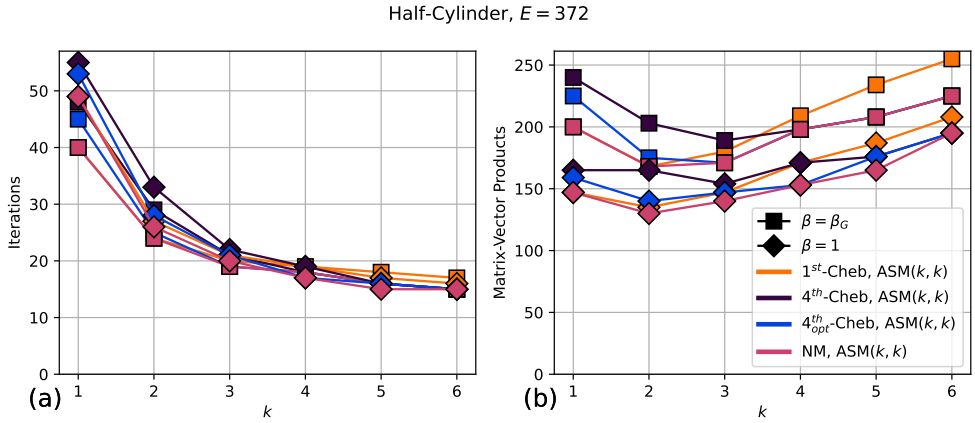



Figure 5.4: Convergence and matrix-vector product count comparison for the additive V-cycle with $\beta = 1$ and $\beta = \beta_G$, $E = 372$.

Does the fact that the improvement in the iteration count using $\beta = \beta_G$ diminishes with respect to the polynomial order imply that the coarse-grid correction is already orthogonal to the residual post correction? Inspection of the values of $\beta_G$ associated with fig. 5.3 indicate

that the value of $\beta_G$ varies greatly for each *iteration* during the solve [6]. To further complicate the matter, there is not a clear trend in the values of $\beta_G$. This, unfortunately, makes *pre-tuning* the $\beta$ coefficient using a one-dimensional optimizer have diminishing returns.

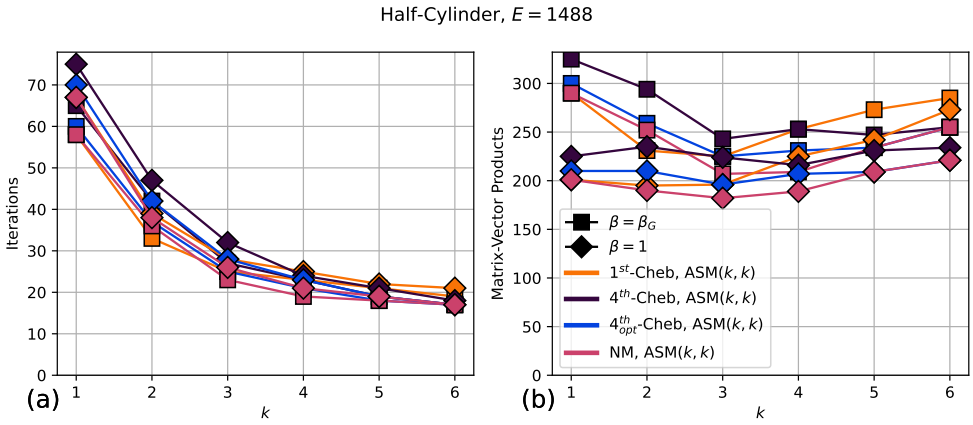



Figure 5.5: Convergence and matrix-vector product count comparison for the additive V-cycle with $\beta = 1$ and $\beta = \beta_G$, $E = 1488$.

Another important question is the optimal smoother polynomial order, $k$. As observed in fig. 5.6, for example, the iteration count decreases from nearly 70 iterations at $k = 1$ to fewer than 20 iterations at $k = 6$. Despite the reduction in the iteration count, the number of matrix-vector products required to reach convergence increases from roughly 175 to nearly 200. In terms of matrix-vector products, the optimal smoother polynomial order for the cases considered here is at $k = 2$ or $k = 3$. We observe similar trends for the multiplicative V-cycle, as shown in section 5.3.2.

### 5.3.2 Comparison Against Multiplicative V-Cycle

As we observed in section 5.3.1, using $\beta = \beta_G$ *improves* the convergence rate of the additive multigrid V-cycle. However, the improvement in the iteration count is not sufficient to make up for the additional work required to compute the correction factor. In this following section, we consider the performance of the additive multigrid V-cycle with $\beta = 1$ compared against the multiplicative V-cycle. As the metrics considered herein only include the iteration count and *fine-level* matrix-vector products required to achieve convergence, the impact of the coarse-grid solve on the solve cost is not factored in here. Despite this

[6] The fact that $\beta_G$ varied *per iteration* is an example of why it is important to consider a *flexible* Krylov subspace method, such as those discussed in section 2.2.

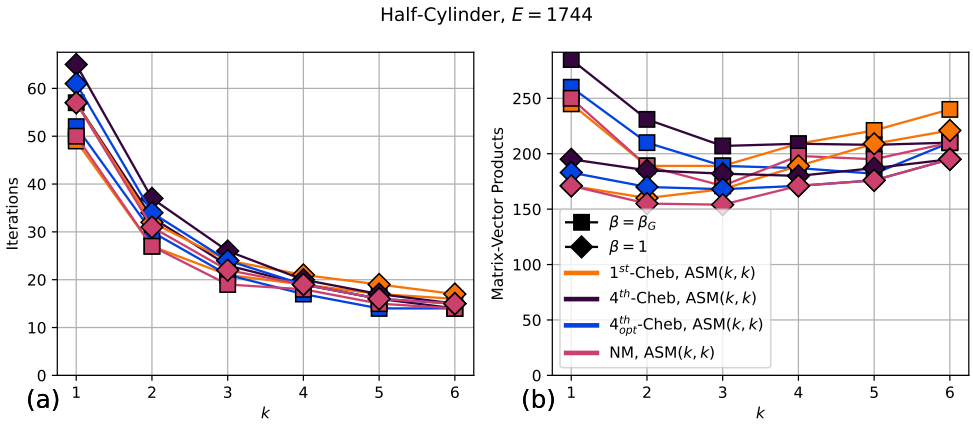



Figure 5.6: Convergence and matrix-vector product count comparison for the additive V-cycle with $\beta = 1$ and $\beta = \beta_G$, $E = 1744$.

limitation, however, the results presented here provide insight into the expected convergence degradation incurred from the additive V-cycle method.

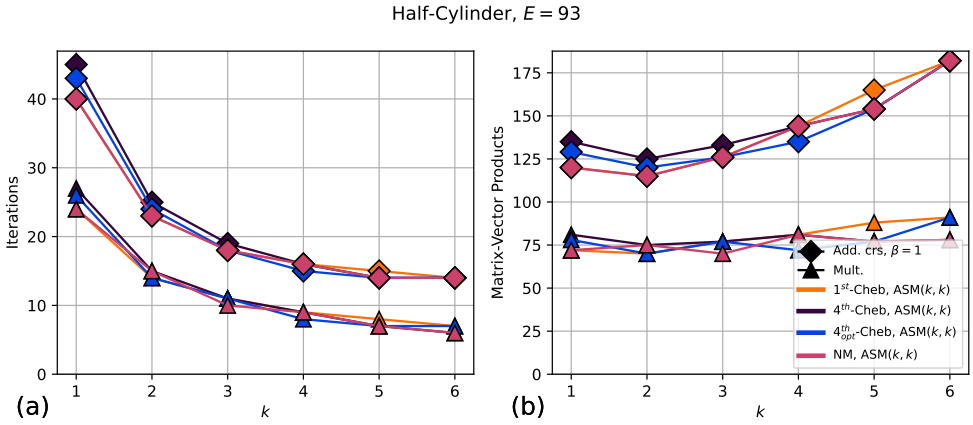



Figure 5.7: Convergence and *fine-level* matrix-vector product count comparison between multiplicative and additive V-cycle with $\beta = 1$, $E = 93$.

Results similar to figs. 5.3 to 5.6 are shown in figs. 5.7 to 5.10, where the additive V-cycle with $\beta = 1$ is compared against the multiplicative V-cycle. As predicted by the condition number bound from Notay and Napov [94] in eq. (5.9), the iteration count associated with the additive multigrid scheme is generally no worse than a factor of two greater than the multiplicative V-cycle. Timing results for the large cases in section 2.8, however, are needed to determine if the additive multigrid V-cycle approach is viable.

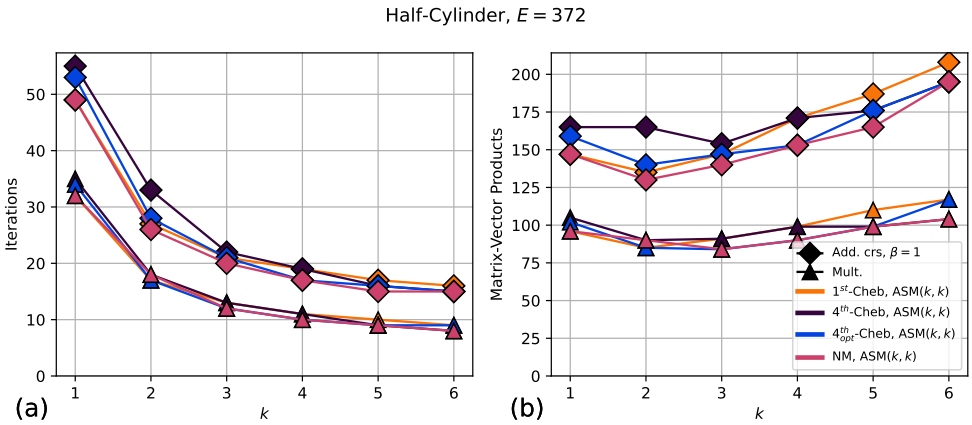



Figure 5.8: Convergence and *fine-level* matrix-vector product count comparison between multiplicative and additive V-cycle with $\beta = 1$, $E = 372$.

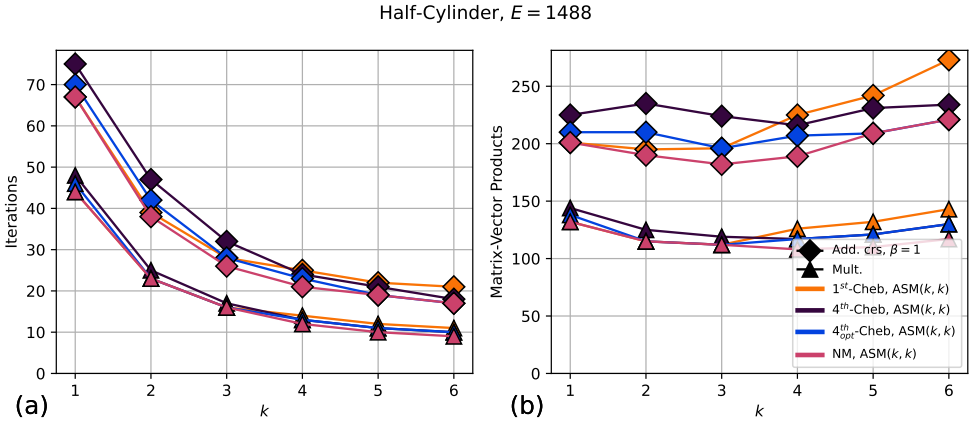



Figure 5.9: Convergence and *fine-level* matrix-vector product count comparison between multiplicative and additive V-cycle with $\beta = 1$, $E = 1488$.

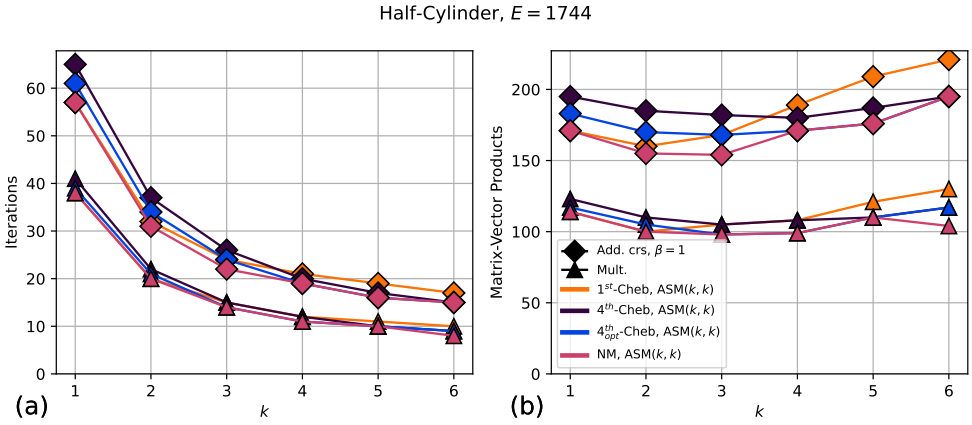



Figure 5.10: Convergence and *fine-level* matrix-vector product count comparison between multiplicative and additive V-cycle with $\beta = 1$, $E = 1744$.

# Chapter 6: Improving Low-Order Refined Preconditioning: Auxiliary Space Methods

The spectral equivalence between the operator for the high-order discretization, $A$, and the low-order discretization, $A_F$, provides a preconditioner with bounded condition number independent of the choice of $E$ and $N$, as discussed in section 2.4. Directly applying $A_F^{-1}$, however, is expensive. In practice, therefore, the action of $A_F^{-1}$ is approximated using algebraic multigrid (AMG), which we denote $M_F^{-1}$. As noted by eq. (2.15), this scheme incurs an additional factor $\kappa(M_F^{-1}A_F)$ to the condition number bound. Further, as noted in section 3.3, the fast application of $M_F^{-1}$ relies on having highly scalable GPU AMG solvers, such as boomerAMG [43, 81], AmgX [42], or MueLu [19].

*What if we could improve the conditioning of the low-order preconditioner?* Assuming the $\kappa(A_F^{-1}A) \sim \pi^2/4$ condition number scaling noted by Orszag [6] for certain cases and taking $M_F^{-1} = A_F^{-1}$, what is the largest improvement that can be achieved in terms of iteration count? Recall the convergence rate of PCG from section 2.2.1 is given as [31]:

$$\|\underline{e}_k\|_A \leq 2\left[\frac{\sqrt{\kappa(M^{-1}A)}-1}{\sqrt{\kappa(M^{-1}A)}+1}\right]^k \|\underline{e}_0\|_A. \tag{6.1}$$

From eq. (6.1), the number of iterations required to reduce the error in the $A$-norm by a factor of $10^8$ is 13. As noted in the preceding paragraph, the condition number scaling suggested by Orszag is not reached in practice, as the action of $A_F^{-1}$ is approximated through AMG, incurring an additional factor $\kappa(M_F^{-1}A_F)$ to the condition number bound in eq. (2.15). Further, if given a preconditioner $M^{-1}$ with $\kappa(M^{-1}A) \sim 1$, then the number of iterations reduces to only one. Achieving this reduction in the condition number is the subject of the remainder of this chapter. Section 6.2 explains the use of the low-order operator as a coarse space for a "two-level" preconditioning scheme, which can lead to an improvement in the condition number. First, however, section 6.1 includes a brief introduction into auxiliary space methods to show that the high-order smoothed SEMFEM (HOS-SEMFEM) method in section 6.2 can be viewed as a special case of an auxiliary space method. Small numerical examples are considered in section 6.3 to demonstrate the efficacy of this approach. However, to enable comparison with the other methods considered in this thesis, larger numerical examples from section 2.8 are deferred to chapter 7.

## 6.1 AUXILIARY SPACE METHODS

In this section, we briefly note the rich history of the use of auxiliary space methods, and even note that the high-order smoothed SEMFEM (HOS-SEMFEM) method proposed in this section can be traced back to 1996 [24]. The fictitious space lemma 6.1 was first formulated by Nepomnyaschikh [99]. This was later extended by Xu [24] to develop the abstract theory for auxiliary space methods to construct non-nested two-level preconditioning techniques comprised of a relaxation scheme (smoother) and an auxiliary space (roughly, a non-nested coarse-grid space). In [24], Xu also proposed utilizing the scheme as a nested two-level method wherein the auxiliary space for a high-order discretization is a low-order discretization, similar to the method proposed in this chapter. Hiptmair and Xu [100] later extended the auxiliary space method to precondition $H(div)$ and $H(curl)$ finite element spaces, without the need to generate a different mesh for the auxiliary space. These techniques were further implemented by Kolev and coworkers as the auxiliary space Maxwell solver (AMS) for $H(curl)$ [101, 102] and auxiliary space divergence solver (ADS) [103] as part of `hypre` [17]. These were further extended to high-order $H(curl)$ [104] to $H(curl)$ and $H(div)$ GPU preconditioners for the entire de Rham complex [15]. In this thesis, however, we focus on the use of auxiliary space methods to develop a nested, multiplicative two-level preconditioner. The low-order discretization is used as the auxiliary space for the high-order discretization. Most notably, the cardinality of the auxiliary space considered in section 6.2 is the same as the cardinality of the high-order discretization. The only difference, then, is the finite element space used to construct the auxiliary space.

In understanding auxiliary space methods, the fictitious space lemma 6.1 is an important result. The fictitious space lemma 6.1, first formulated by Nepomnyaschikh [99] is summarized below as it appears in [105, Theorem 6.3].

**Lemma 6.1.** Let $H, \tilde{H}$ be two real Hilbert spaces equipped with norms induced by $A : H \to H^*$ and $\tilde{A} : \tilde{H} \to \tilde{H}^*$ and let there exist a surjective linear operator $\Pi : \tilde{H} \to H$ such that

$$\|\Pi\tilde{v}\|_A^2 \leq c_0 \|\tilde{v}\|_{\tilde{A}}^2 \qquad \forall \tilde{v} \in \tilde{H} \tag{6.2}$$

and,

$$\forall v \in H \ \exists \tilde{v} \in \tilde{H} \quad \text{such that} \quad v = \Pi\tilde{v} \quad \text{and} \quad \|\tilde{v}\|_{\tilde{A}}^2 \leq c_1 \|v\|_A^2, \tag{6.3}$$

hold. Then, for the preconditioner $\hat{A}_{\mathrm{a}}$ defined by $\hat{A}_{\mathrm{a}}^{-1} := \Pi\tilde{A}^{-1}\Pi^*$, there exists the following

spectral equivalency

$$c_0^{-1}(v,v)_A \leq (\hat{A}_{\text{a}}^{-1}Av,v)_A \leq c_1(v,v)_A \quad \forall v \in H. \tag{6.4}$$

As a result of lemma 6.1, the following condition number bound holds:

$$\kappa(\hat{A}_{\text{a}}^{-1}A) \leq c_0 c_1. \tag{6.5}$$

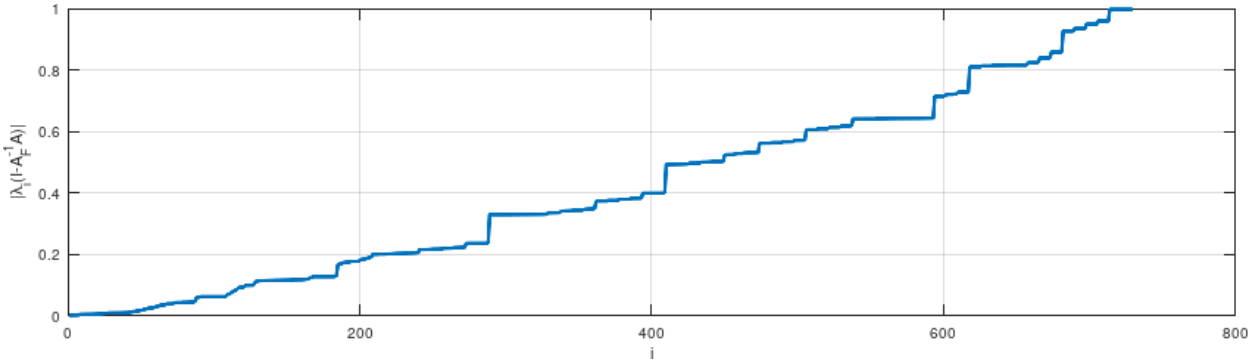
In the view of lemma 6.1, we can view the low-order preconditioner described in section 2.4 as an auxiliary space method. Choosing $\Pi = I$ and $H, \tilde{H}$ to be the Hilbert spaces given by the high-order and low-order discretizations, respectively, we see that lemma 6.1 holds with $c_0, c_1$ given by the high-order/low-order spectral equivalence. Further, the result in eq. (6.5) further predicts the same condition number bound from section 2.4.

## 6.2 LOW-ORDER OPERATOR AS COARSE-GRID SPACE

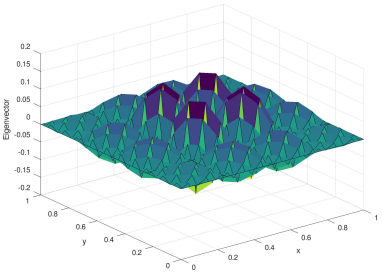
As noted in the results from section 2.9, the low-order discretization can be utilized as the coarse-grid space in a HOS-SEMFEM preconditioning strategy. To better understand this point, consider solving the 2D Poisson equation problem on $\Omega := [0,1]^2$ with $u = 0|_{\partial\Omega}$ and $(E_x, E_y, p) = (4, 4, 7)$. Similar to [14], $\mathbb{P}_1$ finite elements with one triangle per vertex are used to construct $A_F$, which is solved exactly. Eigenvalues and the eigenvectors associated with the three largest (smallest) magnitude eigenvalues of the the error from the preconditioned operator, $I - A_F^{-1}A$, are shown in fig. 6.1. Eigenvectors associated with the largest magnitude eigenvalues of $I - A_F^{-1}A$ (figs. 6.1b to 6.1d) display high-frequency oscillations. At the same time, the eigenvectors associated with the smallest magnitude eigenvalues of $I - A_F^{-1}A$ (figs. 6.1e to 6.1g) are comparatively smooth. As such, $I - A_F^{-1}A$ is able to effectively control long, low-frequency modes. The remaining content that contributes to the condition number of the low-order preconditioner, however, exhibits high-frequency oscillations. How can this observation be used to improve the convergence of preconditioning by $A_F^{-1}$?

Let us recall the two-grid exposition from section 2.6. In section 2.9, we briefly considered a two-grid approach where the SEMFEM discretization is used as a coarse grid system at a lower polynomial order. Mathematically, this corresponds to

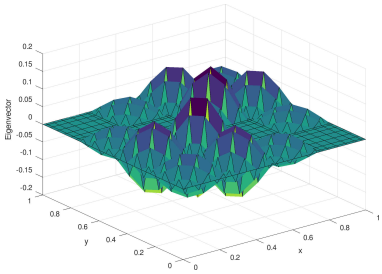
$$\begin{aligned} E_{PTG} &= I - M_{PTG}^{-1}A \\ &= G'(I - \omega_c P A_{c,F}^{-1} P^T A)G, \end{aligned} \tag{6.6}$$



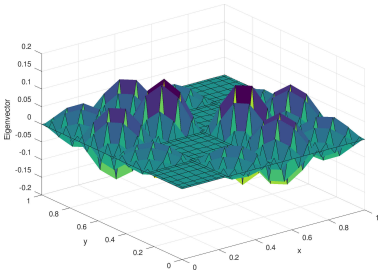

(a) Eigenvalue distribution



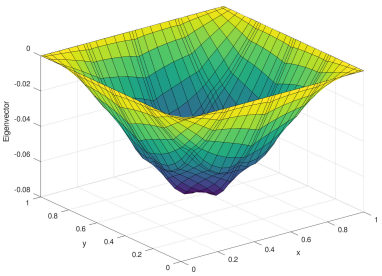
(b) Eigenvector for largest magnitude eigenvalue



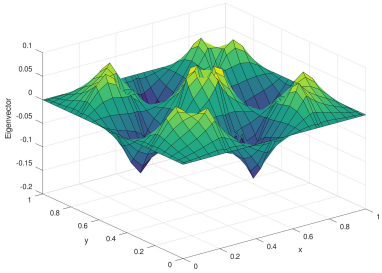
(c) Eigenvector for second largest magnitude eigenvalue



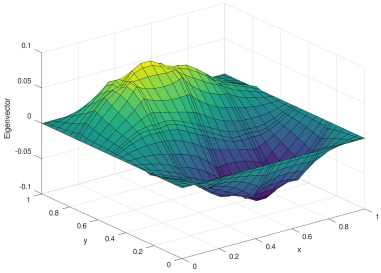
(d) Eigenvector for third largest magnitude eigenvalue



(e) Eigenvector for smallest magnitude eigenvalue



(f) Eigenvector for second smallest magnitude eigenvalue



(g) Eigenvector for third smallest magnitude eigenvalue

Figure 6.1: Eigenvalues/vectors of $I - A_F^{-1}A$ for $p = 7$, $E_x = E_y = 4$ model problem with $\mathbb{P}_1$ finite element space.

where $A_{c,F}$ is the SEMFEM system at a lower polynomial order. In practice, the action of $A_{c,F}^{-1}$ is approximated through preconditioner $M_{c,F}^{-1}$, such as a single algebraic multigrid V-cycle. $G, G'$ are the pre- and post-smoothing operations, respectively. For a more detailed discussion, see sections 2.6, 2.7 and 4.1. This approach is summarized in alg. 6.1, and is the two-level approach described in section 2.9.

Algorithm 6.1 and eq. (6.6) includes an additional *ad-hoc* coarse-grid relaxation parameter, $\omega_c$. This tune-able parameter is included to improve the convergence of the two-grid method. In addition, the introduction of $\omega_c$ addresses the potential volume-scaling defi-

**Algorithm 6.1:** Two-level SEMFEM Scheme

**Data:** Initial guess $\underline{x}$, right-hand side $\underline{b}$
**Result:** Preconditioned solution, $\underline{z}$
$\underline{x} \leftarrow$ `pre-smooth`$(\underline{x}, \underline{b})$ `/* e.g., algs. 2.12 and 4.1 */`
$\underline{r} \leftarrow \underline{b} - A\underline{x}$ `/* Residual re-evaluation */`
$\underline{x} \leftarrow \underline{x} + \omega_c P M_{F,c}^{-1} P^T \underline{r}$ `/* SEMFEM coarse-grid correction */`
$\underline{x} \leftarrow$ `post-smooth`$(\underline{x}, \underline{b})$
$\underline{z} \leftarrow \underline{x}$
**return** $\underline{z}$

ciency of the low-order discretization. Consider, for example, that the one-tet-per-vertex finite element discretization described in section 2.4 does not cover the same volume as the high-order finite element discretization. However, when utilizing the low-order discretization as a preconditioner embedded inside a Krylov subspace method, this scaling factor has no effect on the convergence of the method. In the method described in alg. 6.1, however, the low-order discretization is used as a coarse grid system in a two-level scheme, rescaling the operator by $\omega_c$ may be necessary. Tuning for $\omega_c$, moreover, is not a difficult task. The optimization problem is one-dimensional, and can be solved using a simple derivative-free optimization procedure, such as golden section search. In this work, we consider both $\omega_c = 1$ as well as $\omega_c = \omega_c^*$, the optimizer of the average convergence rate determined during setup through golden section search.

The two-level scheme in eq. (6.6) can be viewed as a perturbation of the Galerkin two-grid method with

$$\begin{aligned} E_{TG} &= I - M_{TG}^{-1} A \\ &= G'(I - P A_c^{-1} P^T A) G, \end{aligned} \tag{6.7}$$

where $A_c = P^T A P$. Notay [57] proved the following eigenvalue relationship relating *any* unperturbed and perturbed two-grid preconditioners:

$$\lambda_{max}(M_{PTG}^{-1} A) \leq \lambda_{max}(M_{TG}^{-1} A) \cdot \max\left(\lambda_{max}(K_c^{-1} A_c), 1\right), \tag{6.8}$$

$$\lambda_{min}(M_{PTG}^{-1} A) \geq \lambda_{min}(M_{TG}^{-1} A) \cdot \min\left(\lambda_{min}(K_c^{-1} A_c), 1\right). \tag{6.9}$$

Here, $A_c$ is the Galerkin coarse grid operator, while $K_c$ is the non-Galerkin coarse grid operator considered in the perturbed two-grid preconditioner. Applying $A_c = P^T A P$ and

$K_c^{-1} = \omega_c A_{c,F}^{-1}$ in our case, we obtain

$$\begin{aligned}\kappa(M_{PTG}^{-1}A) &= \frac{\lambda_{max}(M_{PTG}^{-1}A)}{\lambda_{min}(M_{PTG}^{-1}A)} \\ &\leq \frac{\lambda_{max}(M_{TG}^{-1}A)}{\lambda_{min}(M_{TG}^{-1}A)} \cdot \frac{\max\left(\lambda_{max}(K_c^{-1}A_c), 1\right)}{\min\left(\lambda_{min}(K_c^{-1}A_c), 1\right)} \\ &= \kappa(M_{TG}^{-1}A) \cdot \kappa(A_{c,F}^{-1}A_c) \end{aligned} \tag{6.10}$$

assuming that $\lambda_{min}(\omega_c A_{c,F}^{-1}A_c) \leq 1$ and $\lambda_{max}(\omega_c A_F^{-1}A_c) \geq 1$. The $\kappa(A_{c,F}^{-1}A_c)$ term in eq. (6.10), moreover, exhibits the same spectral equivalence independent of $N$ or $E$ as $\kappa(A_F^{-1}A)$. This latter point implies that the condition number bound in eq. (6.10), however, is larger than that of the SEMFEM preconditioner with $\kappa(A_F^{-1}A)$.

Additional issues, moreover, impact this approach. For example, the reduction in time for applying AMG to evaluate $A_{c,F}^{-1}$ compared to $A_F^{-1}$ may not be large enough to justify the use of this approach. For example, benchmarks on the Kershaw case with $(E, p) = (36^3, 7)$ on $P = 6$ V100 GPUs show that only a 6% reduction in time to apply a single boomerAMG V-cycle is achieved with $p = 6$ compared to $p = 7$. This is despite the 59% reduction in the number of degrees of freedom. However, given that the problem is near the strong-scaling limit, reducing the number of degrees of freedom yields diminishing returns in the time to apply AMG. This, combined with the lack of condition number improvement in eq. (6.10), explains why the proposed two-grid approach with SEMFEM as the coarse grid solve was not successful in [25]. Despite the shortcomings of the preconditioner described in alg. 6.1, the remainder of this section explores ways to address these issues.

*Is there still some way to apply SEMFEM as a coarse grid solve, as fig. 6.1 suggests, while also improving on the condition number?* Consider now the extreme case where the coarse grid and fine grid are the same with $P = I$. The error propagator for the unperturbed "two-grid" method, following eq. (6.7), is

$$\begin{aligned} E_{OG} &= I - M_{OG}^{-1}A \\ &= G'(I - PA_c^{-1}P^TA)G \\ &= G'(I - A^{-1}A)G \\ &= \mathbf{0}. \end{aligned} \tag{6.11}$$

The method converges in a single iteration. However, no progress has been made as the action of $A^{-1}$ is still required. While the Galerkin coarse grid operator is $A_c = P^TAP = A$, replacing the coarse grid operator with a non-Galerkin counterpart can be used to avoid the

need to apply $A^{-1}$. What is a natural choice for a non-Galerkin approximation to $A$ that captures the coarse modes? $A_F^{-1}$! Substituting $A_c = \omega_c A_F^{-1}$ into eq. (6.11) yields

$$\begin{aligned} E_{POG} &= I - M_{POG}^{-1} A \\ &= G'(I - \omega_c A_F^{-1} A)G. \end{aligned} \tag{6.12}$$

Repeating the same analysis from eq. (6.10) with $K_c = \omega_c A_F^{-1}$, and noting that $\kappa(M_{OG}^{-1}A) = 1$, we see that $\kappa(M_{POG}^{-1}A) \leq \kappa(A_F^{-1}A)$. This analysis demonstrates that, at worst, the convergence of the proposed method is no worse than the low-order SEMFEM preconditioner described in section 2.4.

**Algorithm 6.2:** High-order smoothed SEMFEM (HOS-SEMFEM) Scheme

**Data:** Right-hand side, $\underline{b}$
**Result:** Preconditioned solution, $\underline{z}$
$\underline{x} \leftarrow$ `pre-smooth`$(\underline{x}, \underline{b})$ `/* e.g., algs. 2.12 and 4.1 */`
$\underline{r} \leftarrow \underline{b} - A\underline{x}$ `/* Residual re-evaluation */`
$\underline{x} \leftarrow \underline{x} + \omega_c M_F^{-1} \underline{r}$ `/* SEMFEM coarse-grid correction */`
$\underline{x} \leftarrow$ `post-smooth`$(\underline{x}, \underline{b})$
$\underline{z} \leftarrow x$
**return** $\underline{z}$

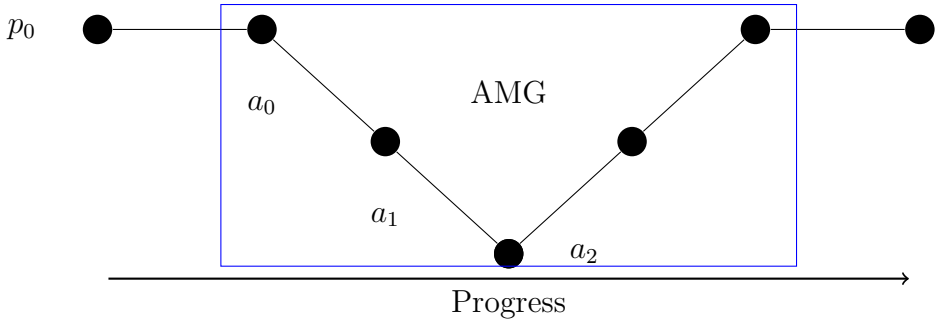



Figure 6.2: Depiction of high-order smoothed SEMFEM preconditioner, described in alg. 6.2.

The HOS-SEMFEM method proposed in the previous paragraph is depicted in fig. 6.2. Algorithm 6.2 is the same as alg. 6.1, except with the interpolator $P$ being replaced with the identity matrix $I$. The inverse of the low-order system is approximated using a single AMG V-cycle, denoted $M_F^{-1}$.

The consideration here of non-Galerkin coarse grid operators is not new. Falgout and coworkers [84] considered constructing non-Galerkin coarse grid operators to improve sparsity. Bienz and coworkers [106] considered sparsification to improve communication by re-

moving weakly connected entries in the coarse operator. Both of these approaches, however, require access to the entries of the operator, and are therefore not amenable for direct use in sparsifying the spectral element operator. At the same time, the proposed approaches in [84, 106] can help in the AMG solve to apply $A_F^{-1}$. A natural choice for a sparsified operator, in this case, is the low-order operator, $A_F$. As discussed in section 2.4, this operator is spectrally equivalent to the high-order operator, and, as shown in fig. 6.1, functions as a good coarse grid operator. Therefore, we propose to use $A_F$ as the coarse grid operator in the "two-level" scheme in eq. (6.12).

The approach mentioned in [25] has been shown to not improve the SEMFEM condition number in eq. (6.10). In addition, the reduction in cost to apply a single AMG V-cycle to approximate $A_{c,F}^{-1}$ is not sufficient to offset any degradation in the convergence rate. The scheme proposed in eq. (6.12), however, is able to improve on the convergence rate at the expense of a few additional matrix-vector products and smoother applications. Therefore, it is expected that the scheme from eq. (6.12) is able to outperform SEMFEM preconditioning in the scenarios where SEMFEM outperforms the pMG-based approaches. It is further expected that this approach can outperform pMG-based approaches under a variety of conditions where the use of SEMFEM preconditioning is not competitive. While most of the results for this method are presented in chapter 7, a few small numerical examples are included in section 6.3 to demonstrate the efficacy of this approach. The following results in section 6.3 and later in chapter 7 directly optimize $\omega_c$ unless otherwise noted. $\omega_c$ is chosen as the minimizer of either the maximum eigenvalue of the error propagator, eq. (6.12), when employing alg. 6.2 as a *solver*; or the average convergence rate for alg. 6.2 as a *preconditioner*.

## 6.3 NUMERICAL EXAMPLES

Numerical examples demonstrating the efficacy of alg. 6.2 are presented in this section. Results from a single 2D spectral element are considered in section 6.3.1. The effect of tuning the coarse-grid relaxation parameter, $\omega_c$, is also considered in section 6.3.1. The half-cylinder cases briefly considered in section 2.4.1 are reconsidered here in section 6.3.2.

### 6.3.1 2D Single Spectral Element

A single spectral element with $p = 9$ is used to discretize the domain $\Omega = [0, 1] \times [0, 4]$. Jacobi smoothing is used to construct the polynomial smoother required in alg. 6.2. Two values of the coarse-grid relaxation parameter, $\omega_c$, are considered. The first is $\omega_c = 1$. The

second, however, solves a one-dimensional optimization problem using golden section search to find the value $\omega_c^*$ that minimizes the max eigenvalue of the error propagator, eq. (6.12). The left plot shows the optimized values of $\omega_c$ for various polynomial smoothers. The right plot shows the maximum eigenvalue of the error propagator for eq. (6.12) various polynomial smoothers. In addition to the suite of polynomial smoothers from section 2.7 and chapter 4, the polynomials directly optimized through Nelder-Mead as presented in section 4.5 are also considered. To observe the effect of optimizing $\omega_c$, the maximum eigenvalue is also plotted for $\omega_c = 1$. Methods that directly optimize $\omega_c$ are denoted by $\omega_c^*$.

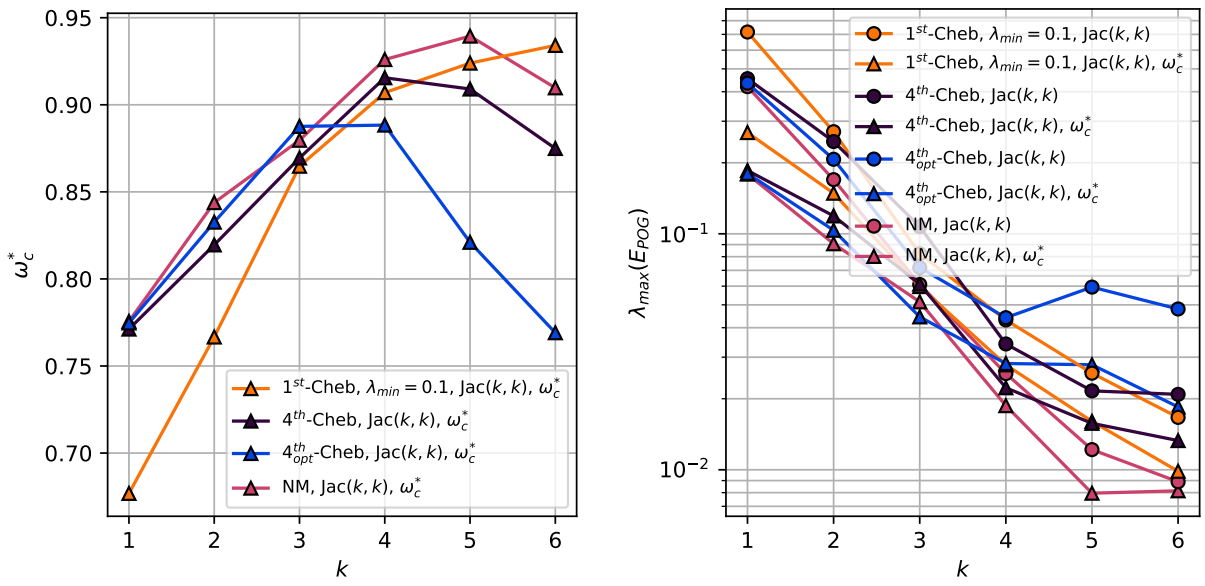



Figure 6.3: Effect of optimizing $\omega_c$ parameter for a single spectral element on $\Omega := [0,1] \times [0,4]$ with $p = 9$. Left: Optimized values of $\omega_c$ for various polynomial smoothers. Right: Max eigenvalue of two-level error propagator for various polynomial smoothers, both with $\omega_c = 1$ and optimized $\omega_c$ values. For simplicity, $\mathbb{Q}_1$ finite elements are used to construct the low-order operator.

In fig. 6.3, the methods which optimize the coarse-grid relaxation parameter, $\omega_c$, significantly reduce the maximum eigenvalue of the error propagator in comparison to the counterparts with $\omega_c = 1$. This indicates that, with a few iterations of golden section search, the coarse-grid relaxation parameter can be tuned to significantly improve the convergence of the two-level method described in alg. 6.2. Since $\omega_c$ can be tuned during setup time with little work, there is no reason not to use the optimized value of $\omega_c$. As such, as noted in section 6.2, we use the optimized value of $\omega_c$ for the remainder of this chapter and the results in chapter 7.

### 6.3.2 Half-Cylinder

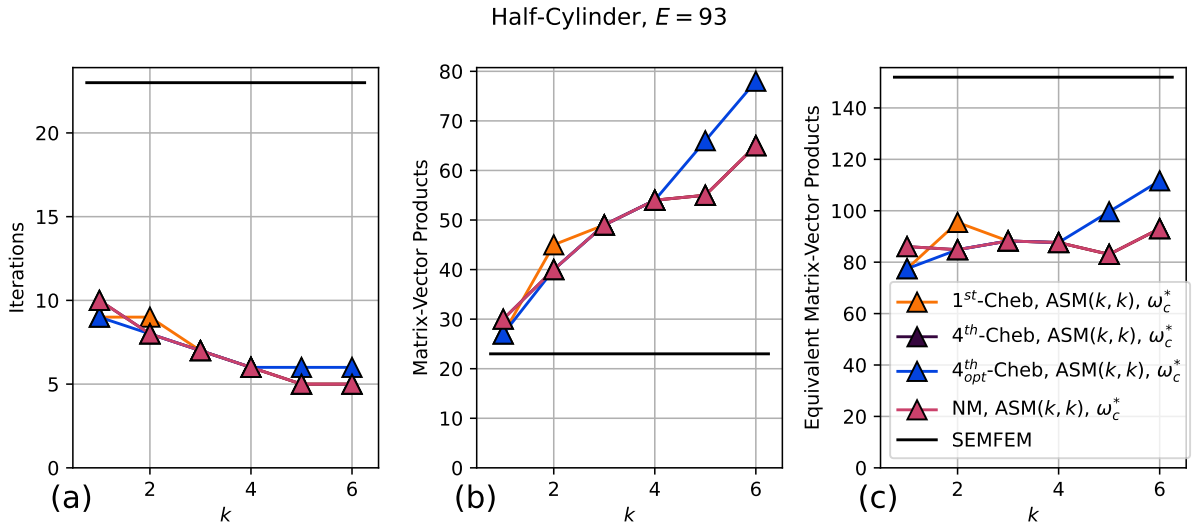



Figure 6.4: Solver performance, half-cylinder, $E = 93$.

The flow over a half-cylinder case from [44], considered in section 2.4.1, is considered here to demonstrate the performance of the HOS-SEMFEM preconditioning scheme described in chapter 6. In each case, the initial residual norm is reduced by a factor of $10^8$. The number of iterations and *fine level* matrix-vector products required to achieve this reduction are reported. In the results below, various polynomial smoothers from section 2.7 and section 4.1 are employed to accelerate the ASM smoother alg. 2.10. Unlike in section 2.4.1, only a single AMG V-cycle is used to approximate the inverse of the low-order matrix. As discussed in chapter 6, an *ad-hoc* relaxation parameter for the coarse-grid correction, $\omega_c$, is directly optimized for a few solver choices. Otherwise, $\omega_c = 1$ is used. The HOS-SEMFEM preconditioner in chapter 6 seeks to balance the relatively expensive AMG V-cycle cost with relatively cheap matrix-vector product and smoother applications. While higher degrees of polynomial smoothing reduces the number of iterations required and the number of AMG V-cycle applications, it also increases the number of matrix-vector products required *per cycle*.

As observed in section 4.4, it is possible that the increase in the polynomial smoother order may decrease the iteration count sufficiently to *reduce* the overall number of matrix-vector products required for the end-to-end solve. Similar to chapter 4, however, much of the cost savings associated with investing in a heavier polynomial smoother come in the form of reducing the iteration count, thereby reducing the number of coarse-grid solve applications. The same is true for the HOS-SEMFEM preconditioner here. A reduction in the iteration

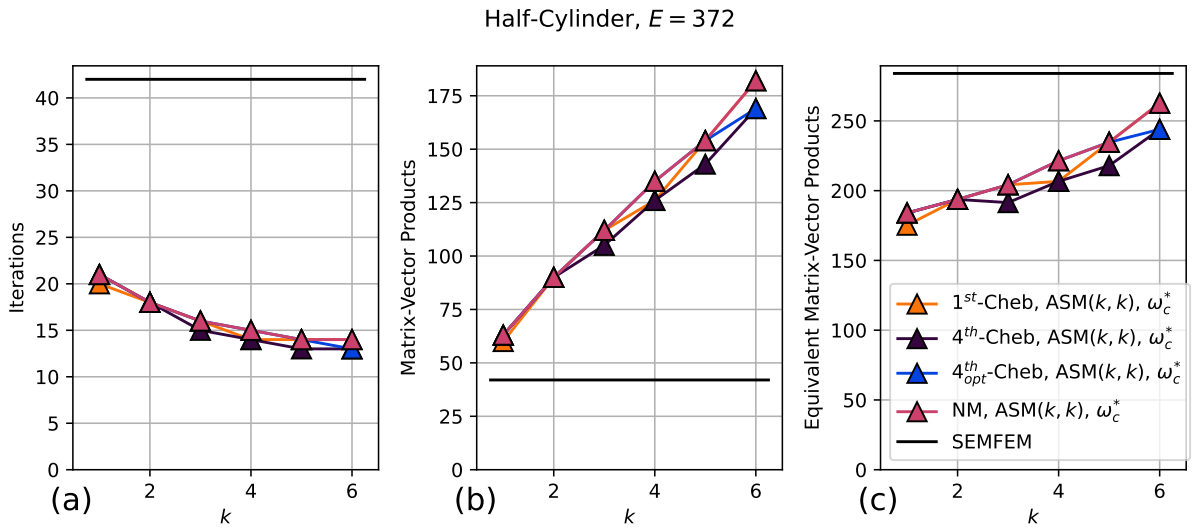



Figure 6.5: Solver performance, half-cylinder, $E = 372$.

count reduces the number of AMG V-cycle calls required to solve the resultant system. While the overall solve time is the most useful metric when comparing the performance trade-off between the potential increase in matrix-vector products and the reduction in the number of AMG V-cycle calls, the timings from the small half-cylinder cases considered here are not representative of the large problems targeting heterogeneous computing platforms.

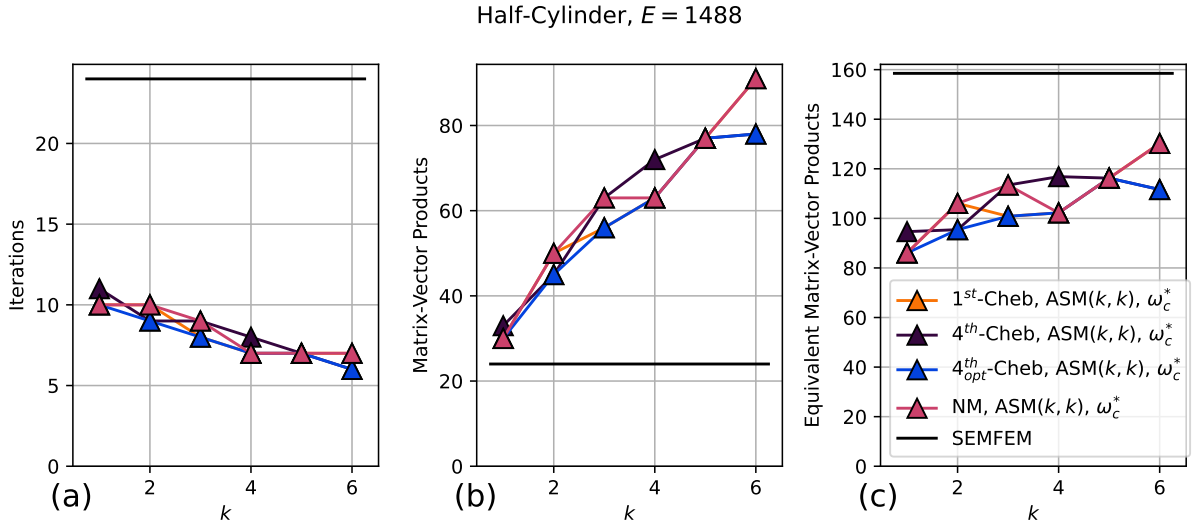



Figure 6.6: Solver performance, half-cylinder, $E = 1488$.

To mitigate the issues discussed in the previous paragraph, we introduce a work measure of the *equivalent matrix-vector products* (eq. (6.13)) to quantify the trade-off between incurring

additional matrix-vector products in the smoother versus reducing the number of AMG V-cycle calls. We first assume that the expense of applying the forward operators $A$ and $A_F$ are equivalent. This is a reasonable assumption based on the throughputs gathered in section 3.2.1 as applying the assembled, sparse matrix $A_F$ has similar throughputs to applying the matrix-free operator, $A$. The cycle complexity of the AMG V-cycle, $C_{AMG}$, is available through boomerAMG as the ratio between the non-zeros in $A_F$ and the non-zeros for all multigrid levels combined. The *equivalent matrix-vector products*, therefore, is

$$\begin{aligned}\text{Equivalent matrix-vector products} &= \text{Matrix-vector products} \\ &+ C_{AMG} \times \text{Iterations} \times \text{AMG smoother passes.}\end{aligned} \tag{6.13}$$

The AMG solver details are explained in section 2.4. Two smoothing passes are employed at each level of the AMG V-cycle. While this metric does not account for the communication costs associated with applying the AMG V-cycle, it does provide a mechanism to measure the relative cost between AMG V-cycle and the matrix-free smoother.

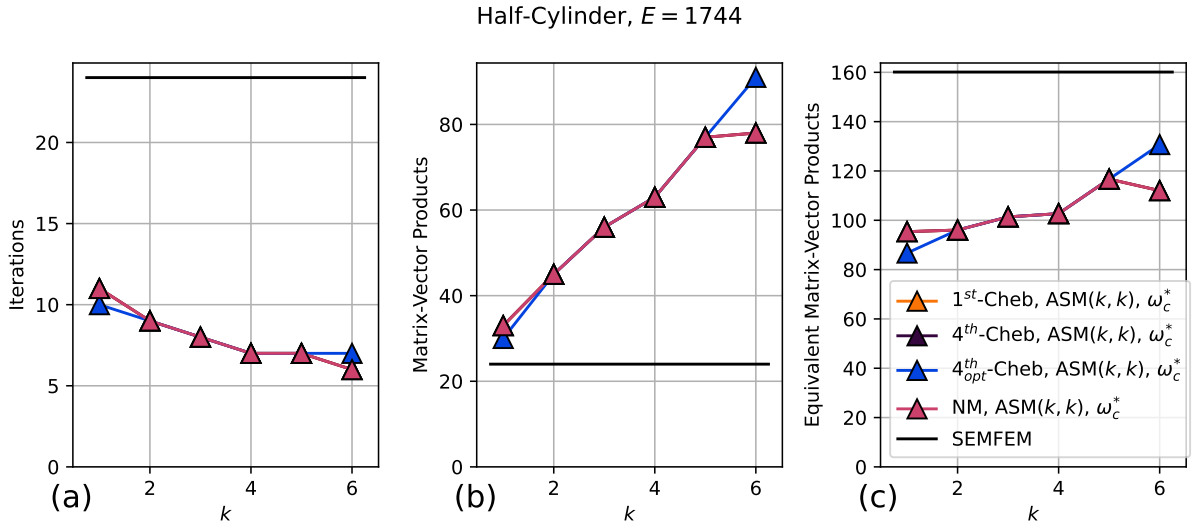
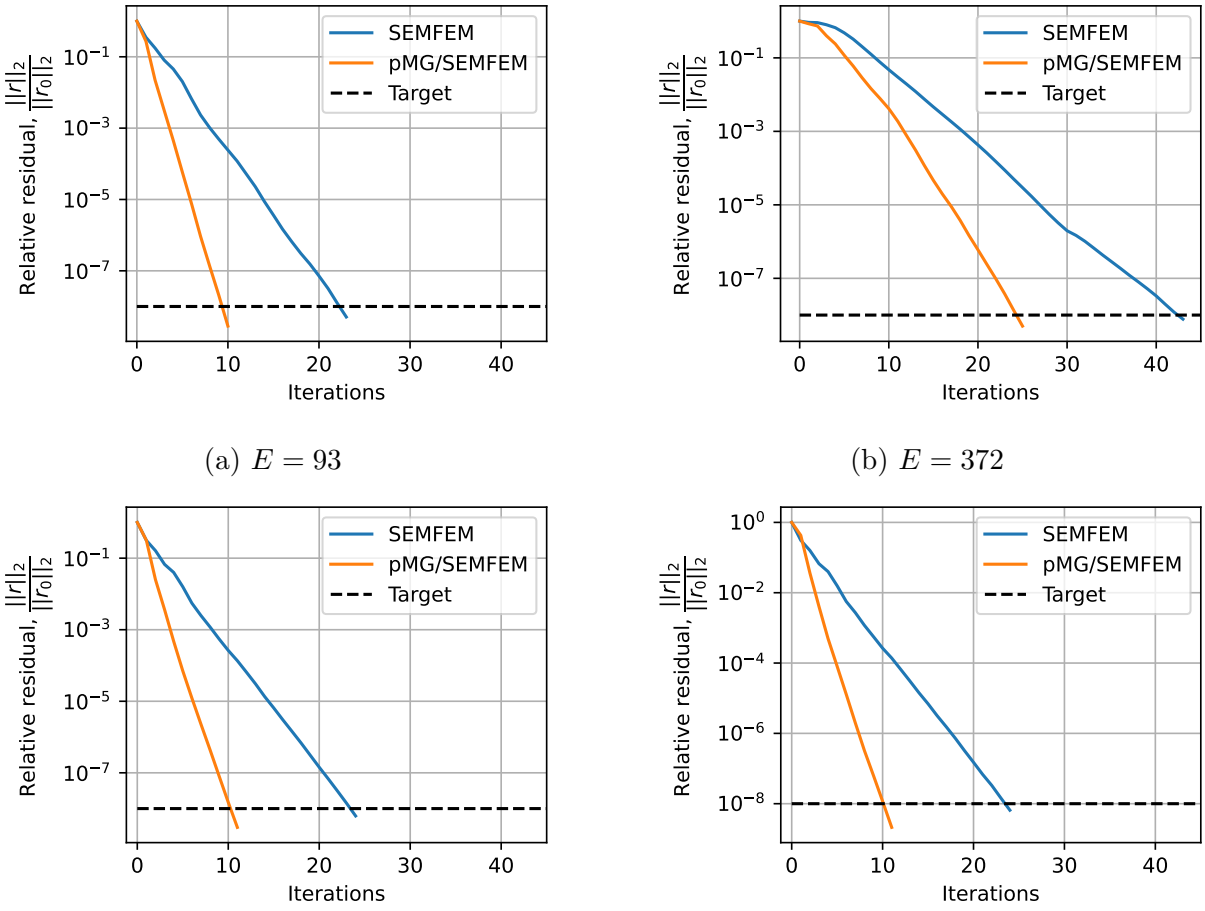



Figure 6.7: Solver performance, half-cylinder, $E = 1744$.

Solver performance results for the half-cylinder case are shown in figs. 6.4 to 6.7. The polynomial smoother order is varied from $k = 1$ to $k = 6$ for a symmetric $(k, k)$ multigrid approach (see chapter 4) utilizing the ASM smoother from section 2.6. We observe that, in all cases, the iteration count is reduced by nearly a factor of two, irrespective of the particular choice of smoother. Despite the increase in the amount of work *per-cycle* employed by the HOS-SEMFEM preconditioner, the number of matrix vector products using a $(1, 1)$-cycle ASM smoother, However, which requires an additional two matrix-vector products per cycle,

is nearly identical to the SEMFEM scheme. However, as the smoother order is increased, the total number of matrix-vector products increases, even though the total number of iterations decreases.




(a) $E = 93$

(b) $E = 372$

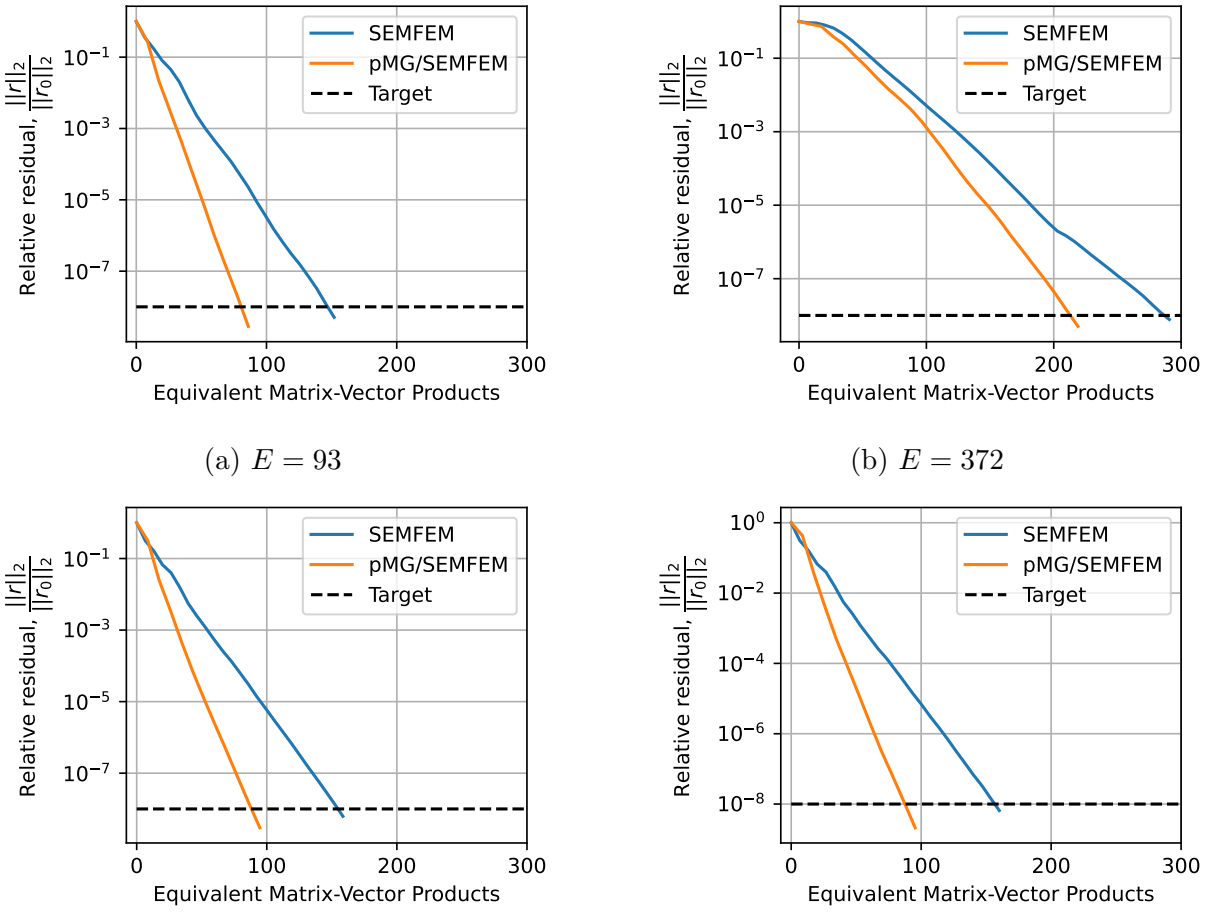
(c) $E = 1488$

(d) $E = 1744$

Figure 6.8: Relative residual history as a function of iteration count for the SEMFEM and best HOS-SEMFEM preconditioner identified in table 6.1.

How does the increase in the matrix-vector products compare to the reduction in the number of AMG V-cycle calls? This is where the equivalent matrix-vector product from eq. (6.13), shown in the rightmost subplot in figs. 6.4 to 6.7, is useful. For the $E = 93$ case (fig. 6.4), the work remains relatively constant with respect to the smoother order, $k$. However, the remaining $E = 372, 1488$, and 1744 cases (figs. 6.5 to 6.7) show that the overall work sharply increases with the smoother order. This would seem to indicate that only a very light smoother should be used in the HOS-SEMFEM preconditioner from alg. 6.2. The equivalent work metric in eq. (6.13), however, does not account for the communication cost

associated with the AMG V-cycle. This cost, especially *at scale*, may be sufficient to justify the use of a higher order polynomial smoother. Results for larger cases, such as those in section 2.8, are deferred to chapter 7 to enable direct comparison with the methods proposed in this thesis.




(a) $E = 93$

(b) $E = 372$

(c) $E = 1488$

(d) $E = 1744$

Figure 6.9: Relative residual history as a function of the *equivalent matrix-vector product* count from eq. (6.13) for the SEMFEM and best HOS-SEMFEM preconditioner identified in table 6.1.

The solvers producing the lowest number of equivalent matrix-vector products (eq. (6.13)) are summarized in table 6.1. Across all of the cases, we observe that ASM smoothing with a first-order polynomial smoother produces the lowest number of equivalent matrix-vector products. The speedup, in terms of equivalent matrix-vector products compared to the SEMFEM preconditioner, is also shown in table 6.1. We observe a range of speedups from 62% for the $E = 372$ case, up to 96% for the $E = 93$ case. The iteration count, as well as the total number of AMG V-cycle calls, is reduced by over a factor of 2 across all cases.

Table 6.1: Half-cylinder solver results. Solver configuration yielding the lowest equivalent matrix-vector product count from eq. (6.13). The iterations, equivalent matrix-vector product counts, for the SEMFEM preconditioner is also reported. The relative *speedup* in terms of equivalent matrix-vector products is also reported.

| $E$ | Solver | Equiv. MV | Equiv. MV, SEMFEM | Speedup | $I$ | $I_{SEMFEM}$ |
|---|---|---|---|---|---|---|
| 93 | $1^{st}$-Cheb, ASM(1,1), $\omega_c^*$ | 77.5 | 152.0 | 1.96 | 9 | 23 |
| 372 | $1^{st}$-Cheb, ASM(1,1), $\omega_c^*$ | 175.3 | 284.0 | 1.62 | 20 | 42 |
| 1488 | $1^{st}$-Cheb, ASM(1,1), $\omega_c^*$ | 86.0 | 158.5 | 1.84 | 10 | 24 |
| 1744 | $4^{th}_{opt}$-Cheb, ASM(1,1), $\omega_c^*$ | 86.7 | 160.1 | 1.85 | 10 | 24 |

Relative residual histories in terms of the iteration count are shown in fig. 6.8. The solver that is optimal with respect to the number of equivalent matrix-vector products from table 6.1 is chosen as the HOS-SEMFEM solver represented in the plot. As the amount of work *per iteration* is different across the HOS-SEMFEM and SEMFEM solvers, the relative residual histories *in terms of equivalent matrix-vector products* are shown in fig. 6.9. We observe that, despite the user-specified solver tolerance, the HOS-SEMFEM solver is able to achieve a lower relative residual than the SEMFEM solver in all cases, *both* in terms of iteration count *and* equivalent matrix-vector products. This makes this method especially attractive, as less *work* is required to achieve a given tolerance and *fewer* AMG V-cycle solves are required. We observe that this trend continues in section 7.2 for the larger cases in section 2.8.

# Chapter 7: Results

So far in this thesis, we have discussed a variety of preconditioner schemes for the solution of the SEM Poisson problem. In chapter 4, we covered different polynomial smoothers and the optimal way of distributing smoothing passes between the pre- and post-smoothing phases of the multigrid V-cycle. Chapter 5 discusses a strategy of overlapping the coarse-grid solve with the remainder of the V-cycle by treating the coarse-grid solve *additively*. Lastly, in chapter 6, we propose an auxiliary space method wherein the low-order operator from section 2.4 is used as a coarse-space combined with smoothing based on the high-order operator. In order to better understand the various scenarios in which a given preconditioner scheme is optimal, we consider the results of the test cases highlighted in section 2.8. All results considered here are on the Oak Ridge Leadership Computing Facility's Summit supercomputer, which is equipped with 2 IBM POWER9 processors and 6 NVIDIA V100 GPUs per node.

The organization of this chapter resembles that of a tournament. The solvers discussed in chapters 2 and 4 to 6 are compared against each other in the battery of tests identified in section 2.8. Section 7.1 characterizes the performance of the multigrid scheme with additive coarse-grid solve discussed in chapter 5. Results for the additive coarse-grid solve, alg. 5.1, are compared against the standard multiplicative V-cycle in alg. 2.8 for the various polynomial smoothers from sections 2.7 and 4.1. The solvers yielding the lowest *time-to-solution* are noted and later compared in section 7.3. In terms of the tournament analogy, the lowest *time-to-solution* solvers advance to the next round. The HOS-SEMFEM approach from alg. 6.2 in chapter 6 is compared against the standard low-order preconditioner from section 2.4. As with section 7.1, the winners advance to the next round. The final round of the tournament, section 7.3, is a comparison of the best solvers from the preceding brackets in sections 7.1 and 7.2. The best solver configuration for each problem is identified. Once known, the best solver configurations from each *class* of methods are compared in a weak scaling study for the Kershaw problem in section 7.3.1.

## 7.1 ADDITIVE COARSE-GRID SOLVE

As we have observed from the scaling results in section 2.9 and fig. 3.19 and the numerical results from section 4.4, the coarse-grid solve component of the geometric $p$-multigrid method in section 2.6 limits the parallel scalability of the solver. By controlling the cost of coarse-grid solves through investing in more *robust* smoothing strategies, we noted in section 4.4,

the effect of the coarse-grid solve on the scalability can be mitigated. Chapter 5, on the other hand, proposes an additive multigrid scheme to overlap the coarse-grid solve with the remainder of the V-cycle. Here, we present numerical results for the cases from section 2.8 to test the efficacy of this approach.

The use of the additive V-cycle approach for the Kershaw problem from section 2.8.1 is considered in section 7.1.1. Section 7.1.2 presents results for the Navier-Stokes pebble bed problems described in section 2.8.2.

### 7.1.1 Kershaw

Here, we consider the additive coarse-grid V-cycle approach from alg. 5.1 for the Kershaw case (section 2.8.1). A single node of summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$ with $E = 36^3$ elements and $p = 7$, the scaling limit from section 2.9. A random right-hand side is generated to tune the NM polynomials from section 4.5. For the linear solve evaluation, the right-hand side is set to the analytical expression eq. (2.33). The solver terminates once a $10^8$ reduction in the residual norm is achieved. The mesh anisotropy factor $\varepsilon = 1, 0.3$ to consider a two distinct mesh anisotropies. The Kershaw case with $\varepsilon = 0.05$ is omitted here, as the solver rarely converges in fewer than 200 iterations. For that particular case, the low-order preconditioning schemes, including the HOS-SEMFEM preconditioner from chapter 6, are more appropriate.

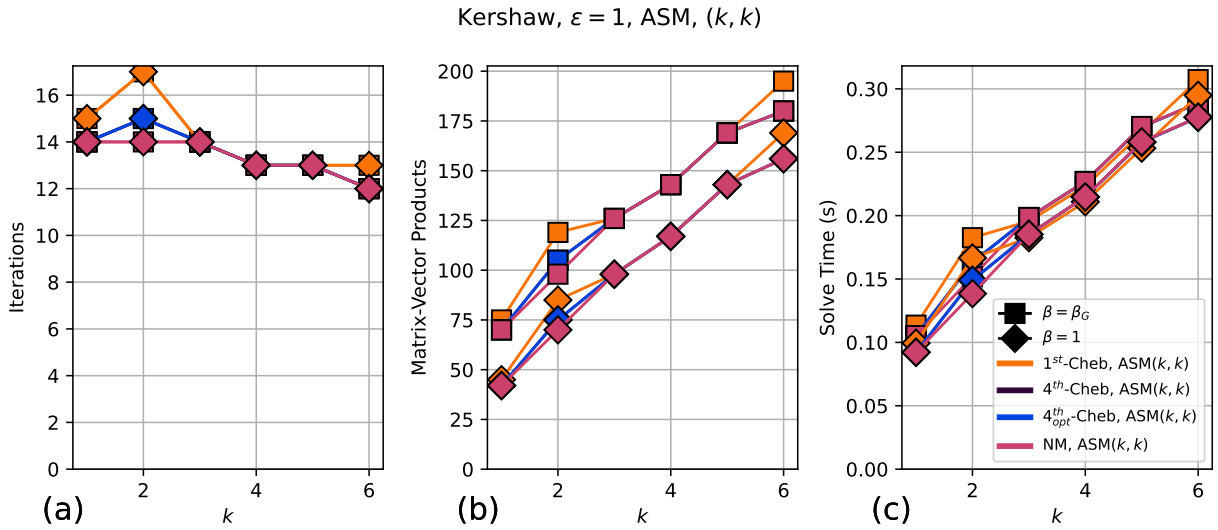



Figure 7.1: Comparison of $\beta = 1$ and $\beta = \beta_G$. Kershaw, $\varepsilon = 1$. $E = 36^3, p = 7$. One Summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. A symmetric distribution of pre- and post-smoothings, $(k, k)$, is used as the smoother polynomial order, $k$, is varied.

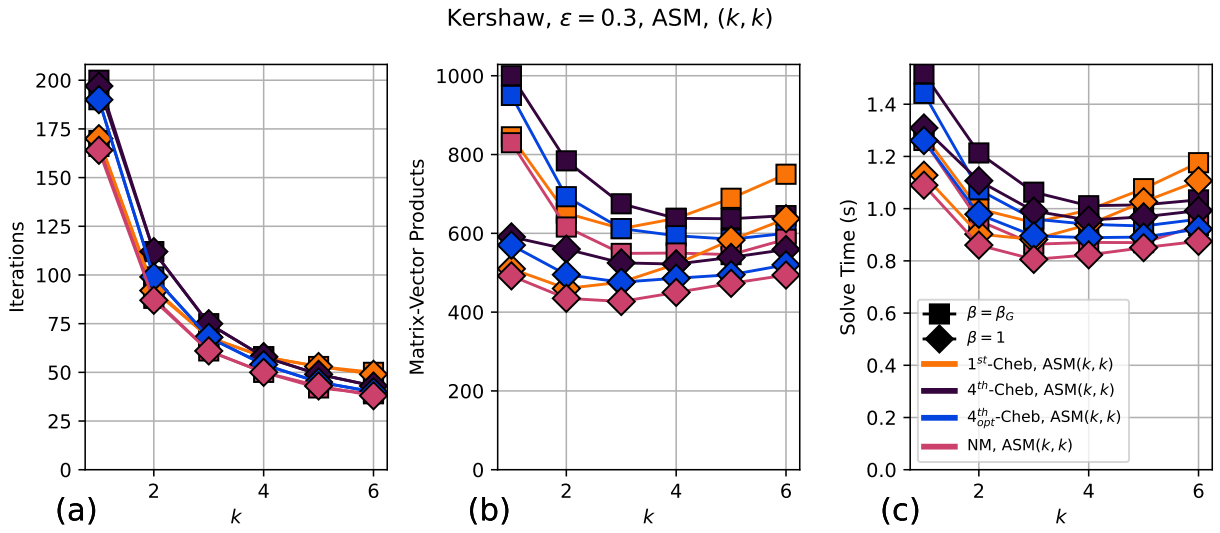



Figure 7.2: Comparison of $\beta = 1$ and $\beta = \beta_G$. Kershaw, $\varepsilon = 0.3$. $E = 36^3, p = 7$. One Summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. A symmetric distribution of pre- and post-smoothings, $(k, k)$, is used as the smoother polynomial order, $k$, is varied.

The first set of results mirror those from section 5.3.1. Here, we consider the use of two distinct values of $\beta$ from alg. 5.1: $\beta = 1$ and the Greenbaum $\beta = \beta_G$ (eq. (5.8)). The former requires no work to compute, while the latter requires two additional matrix-vector products *per cycle*. As we observed in section 5.3.1, $\beta = \beta_G$ tends to *improve* the convergence rate of the solver. However, the added cost is not justified by the reduction in the iteration count. In the following results, we demonstrate that $\beta = \beta_G$ has virtually no effect on the convergence rate of the solver.

Iteration count, matrix-vector product count, and time-to-solution for the Kershaw cases are shown in figs. 7.1 and 7.2. As clearly shown, the use of $\beta = \beta_G$ has virtually no effect on the convergence rate of the solver. Unlike in section 5.3.2, inspection of the $\beta_G$ coefficients for the case in fig. 7.1 reveal that $\beta_G \sim 1$. That is, the coarse-grid component is already nearly orthogonal to the residual after correction. This makes the additional work in computing $\beta_G$ redundant.

Further hurting the use of $\beta = \beta_G$, the expense required in the two additional two matrix-vector products *per iteration* is expensive. In fig. 7.2, for example, the matrix-vector product count nearly *doubles* at $k = 1$ or $k = 2$. As the polynomial order increases, however, this additional expense becomes smaller. This is purely from the fact that, at higher smoother polynomial order, the iteration count is reduced. This mitigates the additional two matrix-vector products *per iteration*.

As the results in figs. 7.1 and 7.2 demonstrate there is no benefit to choosing $\beta = \beta_G$,

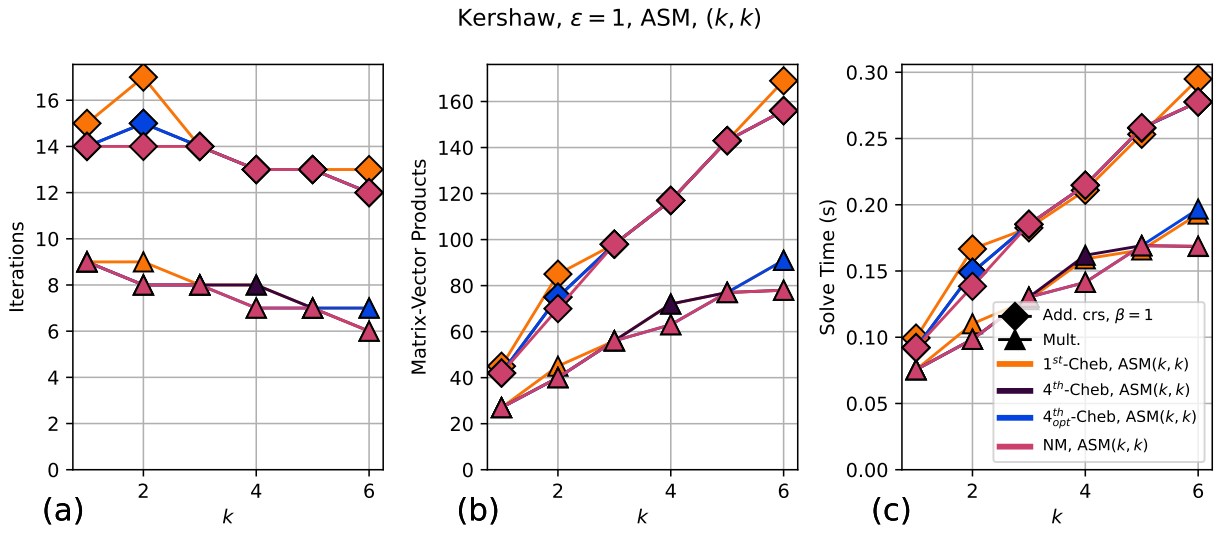



Figure 7.3: Comparison of additive coarse-grid method with $\beta = 1$ the standard, multiplicative pMG method. Kershaw, $\varepsilon = 1$. $E = 36^3, p = 7$. One Summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. A symmetric distribution of pre- and post-smoothings, $(k, k)$, is used as the smoother polynomial order, $k$, is varied.

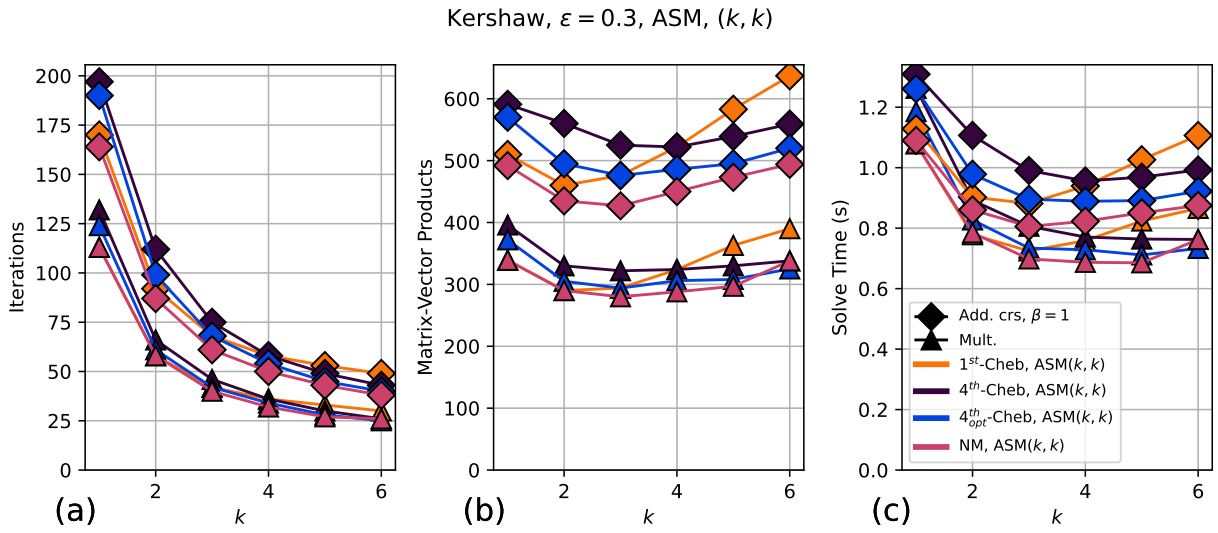



Figure 7.4: Comparison of additive coarse-grid method with $\beta = 1$ the standard, multiplicative pMG method. Kershaw, $\varepsilon = 0.3$. $E = 36^3, p = 7$. One Summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. A symmetric distribution of pre- and post-smoothings, $(k, k)$, is used as the smoother polynomial order, $k$, is varied.

we consider $\beta = 1$ when comparing the performance against the standard, multiplicative pMG approach from section 2.6. Similar to figs. 7.1 and 7.2. iteration count, matrix-vector product count, and the time-to-solution are reported in figs. 7.3 and 7.4.

Figures 7.3 and 7.4 indicates that there are no cases in which the additive approach is able to outperform the multiplicative V-cycle approach. The solver degradation predicted in the two-level convergence bound from eq. (5.9) is too large for the additive approach to overcome. However, in nearly all cases, the additive approach remains within "striking distance" of the multiplicative approach. For this relatively small case with $P = 6$ processors, the relative cost of the coarse-grid solve is minimal. *At scale*, however, this cost becomes more significant. In this regime, therefore, we expect the additive approach to be more competitive than shown here. Further, the tuning strategy suggested as future work in section 5.2 may also improve the performance of the additive approach. For the results shown in figs. 7.3 and 7.4, a *fixed* coarse-grid solve approximation is used. However, especially as the smoother polynomial order increases, the amount of work in the multiplicative portion of the V-cycle increases. Therefore, the solution accuracy of the coarse-grid solve approximation can be improved *at no additional run-time cost* by balancing the relative time of each multigrid component. For example, if the multiplicative portion of the V-cycle takes more time than the coarse-grid solve, the coarse-grid component can be improved through additional AMG smoothings *without increasing the overall run-time per iteration*. This is a significant advantage of the additive approach over the multiplicative approach that is not fully realized in this work.

The results discussed in this subsection utilize the symmetric distribution of pre- and post-smoothing applications, $(k, k)$. For completeness, additional results for the $(2k, 0)$ distribution are shown in appendix B.1.1. These results are qualitatively similar to those shown here.

### 7.1.2 Pebble Cases

The Navier-Stokes pebble cases from section 2.8.2 are considered here. These three cases consist of 146, 1568, and 67 contacting pebbles embedded in a cylinder. To avoid the full time-stepping required for the Navier-Stokes equations, as in the results from section 2.9.2, the right-hand side corresponding to the last pressure Poisson problem is used as the right-hand side here. To tune the NM polynomial smoother in section 4.5, a random right-hand side is generated by taking a random solution vector in $[0, 1]$ that satisfies the pressure boundary conditions and applying the discrete Laplacian operator.

As in section 7.1.1, we start by comparing the performance of the additive coarse-grid solve scheme with $\beta = 1$ and $\beta = \beta_G$. The results are shown in figs. 7.5 to 7.7. Similar to the Kershaw results (section 7.1.1), $\beta = \beta_G$ has the same convergence as $\beta = 1$. The former, however, requires *more work per cycle* to compute the correction. Therefore, in the subsequent results comparing the additive coarse-grid solve with the multiplicative V-cycle,

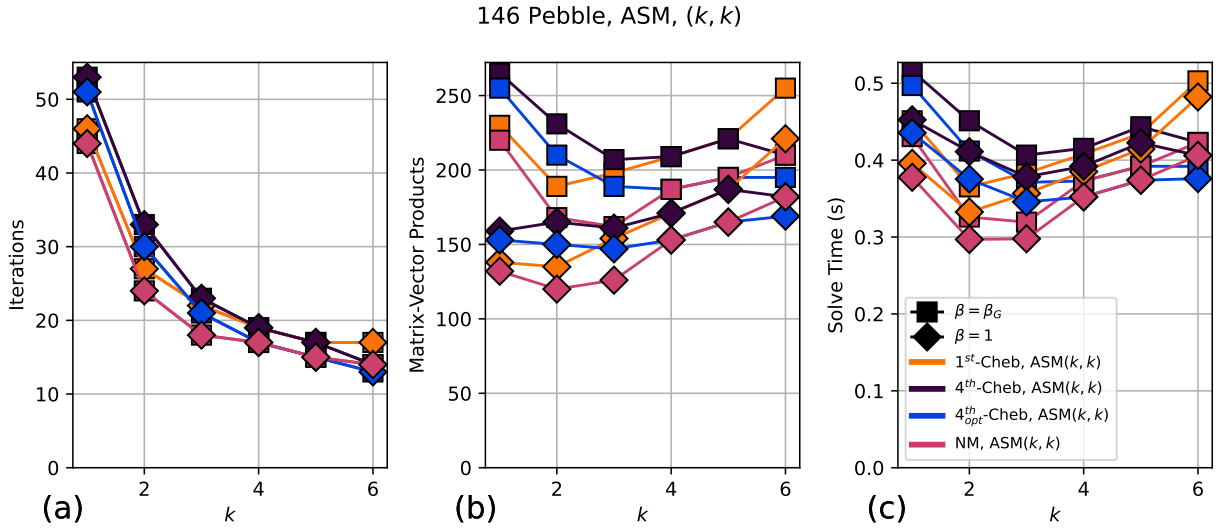



Figure 7.5: 146 pebble case (fig. 2.11a). Comparison of $\beta = 1$ and $\beta = \beta_G$ for the additive coarse-grid method. A single summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 3.5M$. The polynomial smoother order, $k$, is varied. Pre- and post-smoothing are equal, denoted as $(k, k)$.

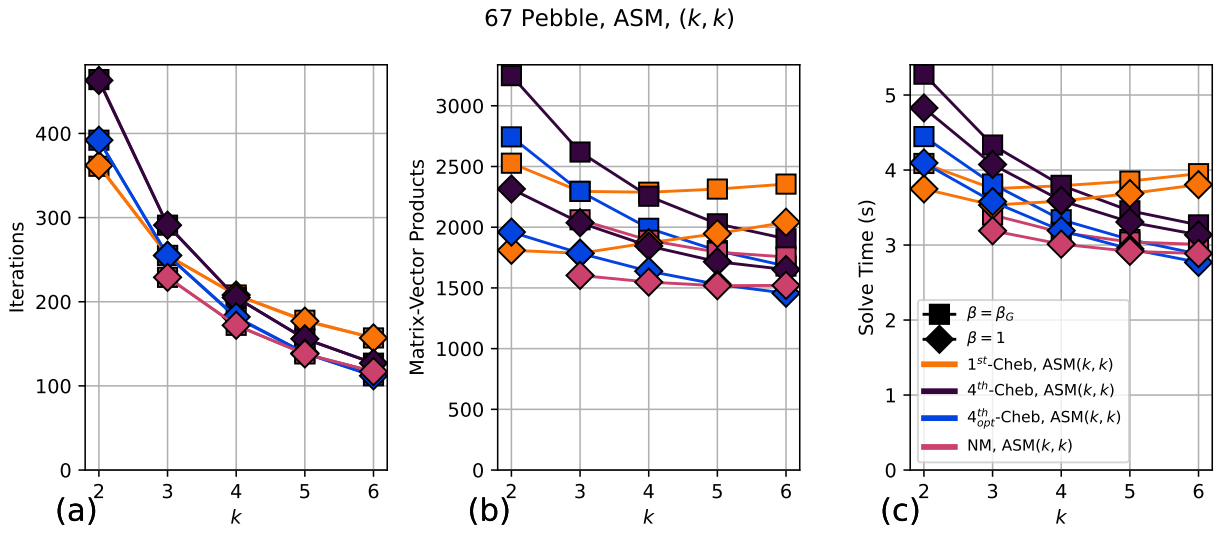



Figure 7.6: 67 pebble case (fig. 2.11c). Comparison of $\beta = 1$ and $\beta = \beta_G$ for the additive coarse-grid method. Three summit nodes ($P = 18$ V100 GPUs) is used with $n/P \sim 2.33M$. The polynomial smoother order, $k$, is varied. Pre- and post-smoothing are equal, denoted as $(k, k)$.

we use $\beta = 1$.

In figs. 7.5 to 7.7, the additive coarse-grid solve with $\beta = 1$ is compared against the same method with $\beta = \beta_G$. Figures 7.8 to 7.10 now compare the additive coarse-grid solve with

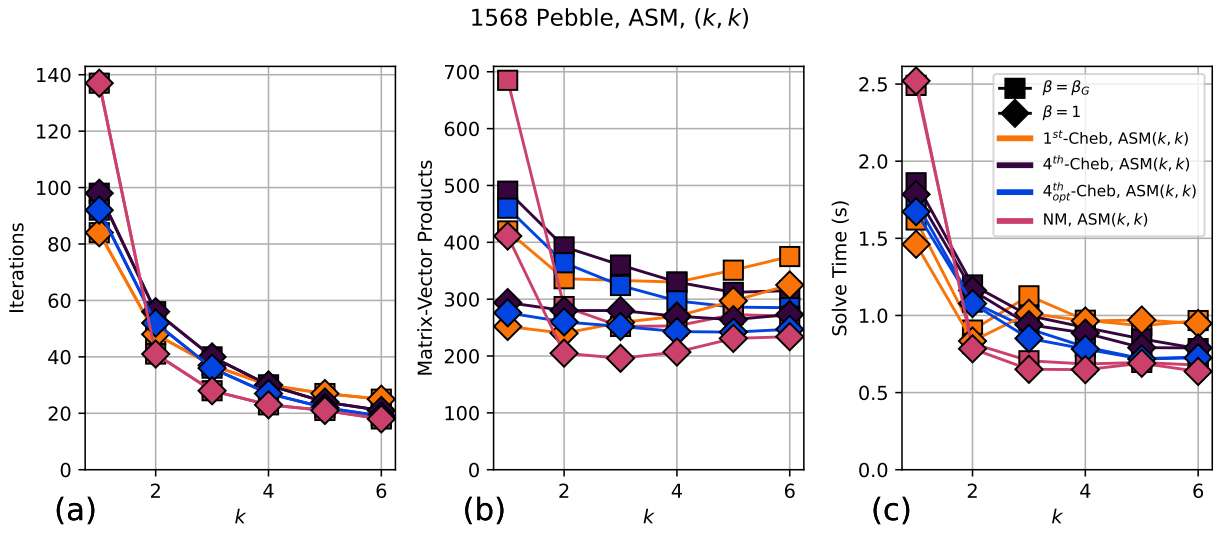



Figure 7.7: 1568 pebble case (fig. 2.11b). Comparison of $\beta = 1$ and $\beta = \beta_G$ for the additive coarse-grid method. 12 summit nodes ($P = 72$ V100 GPUs) is used with $n/P \sim 2.5M$. The polynomial smoother order, $k$, is varied. Pre- and post-smoothing are equal, denoted as $(k, k)$.

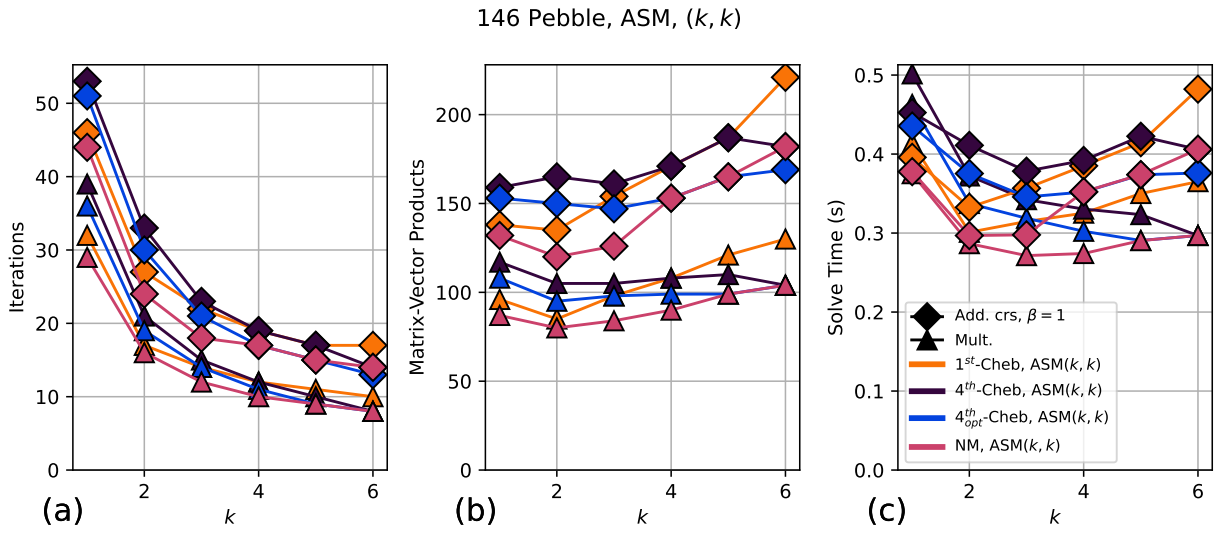



Figure 7.8: 146 pebble case (fig. 2.11a). Comparison of additive coarse-grid solve with $\beta = 1$ and multiplicative V-cycle. A single summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 3.5M$. The polynomial smoother order, $k$, is varied. Pre- and post-smoothing are equal, denoted as $(k, k)$.

$\beta = 1$ to the multiplicative V-cycle approach. The results are similar to those from the Kershaw cases (section 7.1.1). There are no scenarios identified herein where the additive coarse-grid solve method outperforms the multiplicative V-cycle. This unfortunate fact is

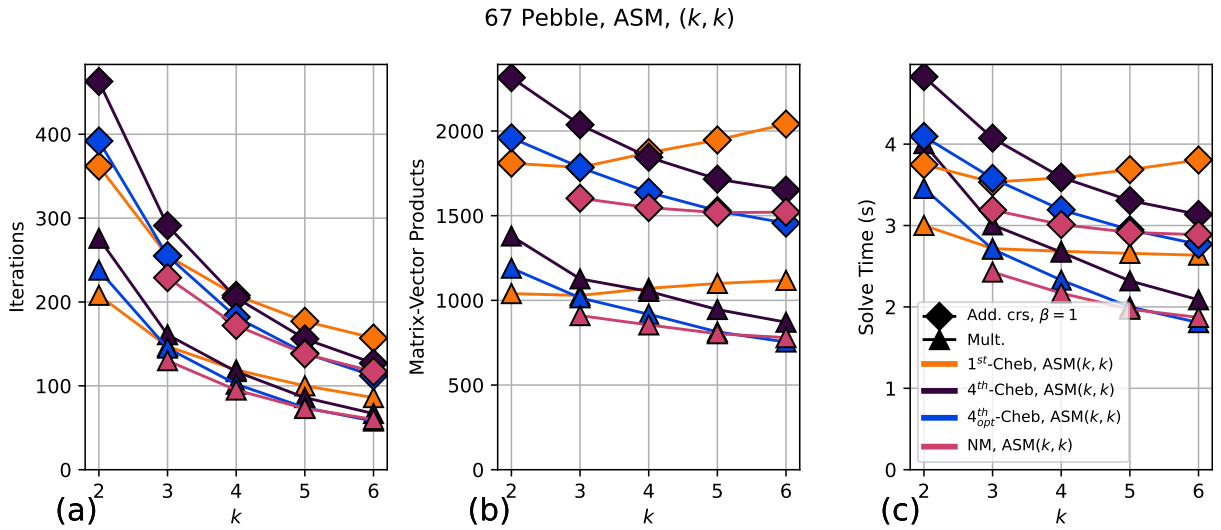



Figure 7.9: 67 pebble case (fig. 2.11c). Comparison of additive coarse-grid solve with $\beta = 1$ and multiplicative V-cycle. Three summit nodes ($P = 18$ V100 GPUs) is used with $n/P \sim 2.33M$. The polynomial smoother order, $k$, is varied. Pre- and post-smoothing are equal, denoted as $(k, k)$.

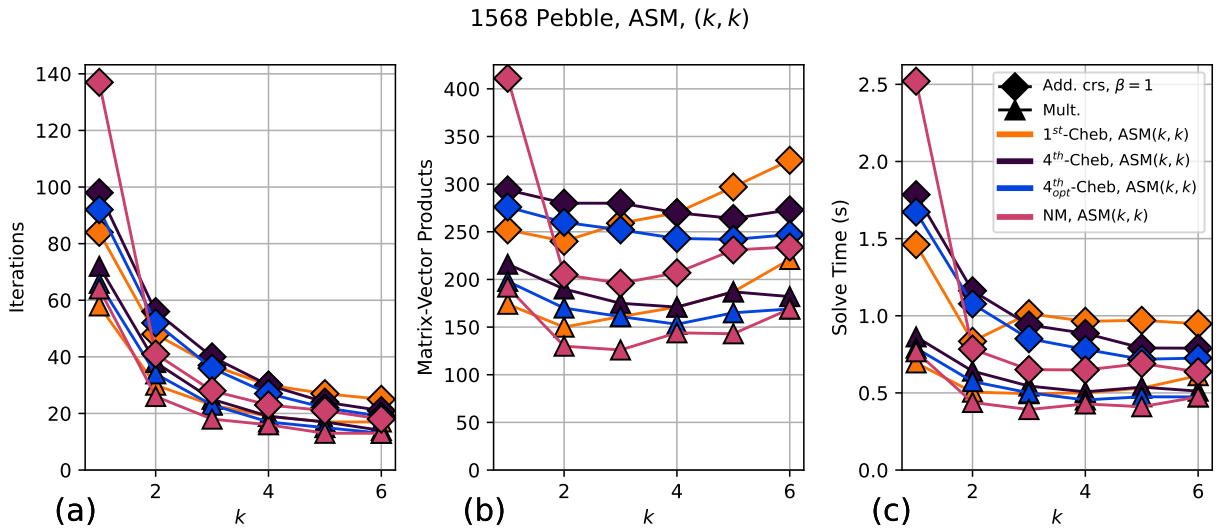



Figure 7.10: 1568 pebble case (fig. 2.11b). Comparison of additive coarse-grid solve with $\beta = 1$ and multiplicative V-cycle. 12 Summit nodes ($P = 72$ V100 GPUs) is used with $n/P \sim 2.5M$. The polynomial smoother order, $k$, is varied. Pre- and post-smoothing are equal, denoted as $(k, k)$.

due to the *increase* in the number of iterations required by the additive approach. The speedup *per iteration* in the preconditioner is not sufficient to overcome the increase in the number of iterations.

The results discussed in this subsection utilize the symmetric distribution of pre- and post-smoothing applications, $(k, k)$. While similar to the results shown in this section, for completeness, additional results for the $(2k, 0)$ distribution are shown in appendix B.1.2.

Table 7.1: Solver configuration yielding the lowest time-to-solution in seconds, $T_s$, comparing the additive multigrid scheme from chapter 5 and the multiplicative V-cycle from section 2.6. ASM (alg. 2.10) smoothing is accelerated via different polynomial smoothers.

| Case | Solver | $T_s$ | Iterations |
|---|---|---|---|
| Kershaw, $\varepsilon = 1$ | NM, ASM(1,1) | 0.08 | 9 |
| Kershaw, $\varepsilon = 0.3$ | NM, ASM(6,0) | 0.65 | 43 |
| 146 pebble | NM, ASM(3,3) | 0.27 | 12 |
| 67 pebble | $4^{th}_{opt}$-Cheb, ASM(12,0) | 1.53 | 49 |
| 1568 pebble | NM, ASM(6,0) | 0.39 | 18 |

The results presented in sections 7.1.1 and 7.1.2 are summarized in table 7.2 are summarized in table 7.1. Here, the solver achieving the lowest *time-to-solution* for each case is highlighted across the additive coarse-grid and multiplicative V-cycle approaches. As discussed in sections 7.1.1 and 7.1.2, the additive multigrid scheme is unable to outperform the standard multiplicative V-cycle. Part of the reason *why* the additive method in chapter 5 cannot outperform the multiplicative V-cycle is that the coarse-grid solve cost is already effectively *controlled* by the robust polynomial smoothers. In this regime, the portion of the simulation that is spent in the coarse-grid solve is already well controlled, assuming that a reasonable smoother order is chosen. However, there are scenarios in which the additive coarse-grid solve approach is "in the ball park" of the multiplicative V-cycle. For these cases, the tuning strategy suggested in section 5.2 may be used to further improve the performance of the additive coarse-grid solve approach. This, however, is left as future work for this thesis. When comparing the results against the low-order auxiliary space method described in chapter 6 in section 7.2, the multiplicative V-cycle solvers highlighted in table 7.1 progress to the next round of the solver tournament.

To better understand the relative cost of each operation in the multiplicative and additive V-cycle approaches, we show the ratio between the remainder of the V-cycle to the coarse-grid solve, as well as the time per iteration, in fig. 7.11. We first note that the majority of the solution time is spent in the smoother phases of the V-cycle, as shown in the left subplot in fig. 7.11, except for very-low polynomial orders. As the polynomial order increases, the time spent in the coarse-grid solve *per cycle* decreases, as expected. This indicates that, in the additive V-cycle approach, the solve accuracy of the coarse-grid solve can be increased without impacting the *time per iteration*. Whether this is sufficient to overcome

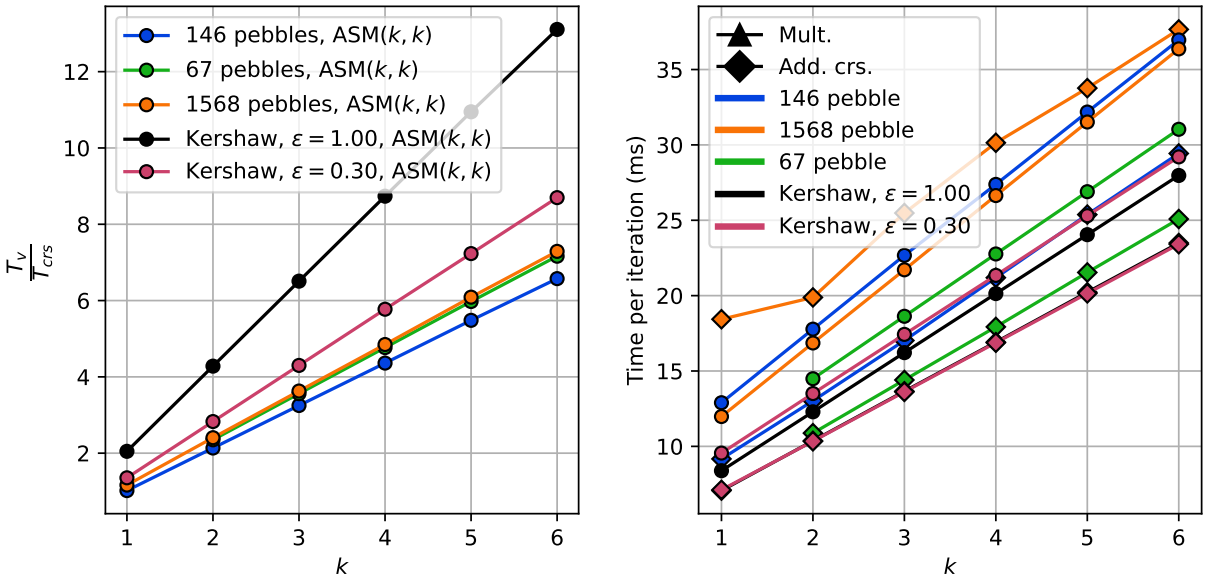



Figure 7.11: Left: ratio of time spent in the remainder of the V-cycle compared to the time spent in the coarse grid solve. Right: time per iteration in milliseconds for the additive and multiplicative V-cycle preconditioner. Note that neither metrics are impacted by the particular choice of polynomial smoother (e.g., first-kind versus fourth-kind Chebyshev) from chapter 4.

the convergence rate degradation from the additive approach, however, is left as future work. With exception to the 1568 pebble case, our expectation that the additive V-cycle yields a cheaper preconditioner *per iteration* is confirmed in the right subplot in fig. 7.11. However, the reduction in the time per iteration is not sufficient to overcome the convergence rate degradation of the additive V-cycle approach.

## 7.2 IMPROVING LOW-ORDER PRECONDITIONERS: AUXILIARY SPACE METHODS

In chapter 6, we discussed the idea of using the low-order finite element equivalence with the high-order SEM operator as a coarse-space in an auxiliary space method. Here, we consider the use of this approach, described in alg. 6.2, as a preconditioner for the high-order SEM operator. This "HOS-SEMFEM" approach seeks to minimize the number of relatively expensive AMG V-cycles associated with the approximation of the inverse of the low-order operator. As seen in section 6.3, this approach is promising as it both reduces the *total work metric* referred to as the *equivalent matrix-vector product* count eq. (6.13) as well as reducing the number of AMG V-cycles required to reach a given tolerance. Here,

we now consider the moderate scale cases from section 2.8 on the Oak Ridge Leadership Computing Facility's Summit supercomputer. For simplicity, we compare against the low-order preconditioning scheme described in section 2.4. Comparison against other methods is deferred to section 7.3. The Kershaw test case form section 2.8.1 is considered in section 7.2.1. Results fro the Navier-Stokes pebble cases from section 2.8.2 are considered in section 7.2.2.

### 7.2.1 Kershaw

The Kershaw case from section 2.8.1 is considered here to test the efficacy of the HOS-SEMFEM preconditioner in alg. 6.2. The problem is discretized using $E = 36^3$ elements with polynomial degree $p = 7$. A single node of summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$, which roughly corresponds to the solver strong-scale limits identified in section 2.9. To account for different levels of geometric deformation in the mesh, three values of $\varepsilon$ are considered: $\varepsilon = 1, 0.3, 0.05$. The right-hand side is set to the analytical expression eq. (2.33). A random solution vector satisfying the boundary condition is used to construct the right-hand side used for tuning the $\omega_c$ parameter, as well as the NM smoother polynomials from section 4.5. The reason for using this alternate right-hand side for tuning is to demonstrate the transferability of the tuned parameters to different right-hand sides.

Algorithm 6.2 is utilized as a preconditioner with GMRES(15) as the Krylov solver. ASM smoothing (alg. 2.10) is accelerated via the various polynomial smoothers considered in sections 2.7 and 4.1. $k$ pre- and post-smoothing passes (denoted as $(k, k)$, see chapter 4) is used for all polynomial smoothers. Results are shown in figs. 7.12 to 7.14.

How does the HOS-SEMFEM scheme from alg. 6.2 compare to the SEMFEM preconditioner from section 2.4? As shown in subplot (a) from figs. 7.12 to 7.14, the HOS-SEMFEM preconditioner yields a sharp decrease in the overall iteration count. For example, with $\varepsilon = 0.3$, the iteration count reduces by nearly a factor of two from 147 to 74 with $k = 1$ ASM smoothing passes. This large reduction in the iteration count at the expense of a few relatively inexpensive smoothing operations yields an appreciable speedup in the time-to-solution, as shown in subplot (d). With $\varepsilon = 0.3$, for example, the time-to-solution drops from roughly 1.8 to 1 second with $k = 2$ smoothing passes, an 80% speedup. While the SEMFEM method is not competitive for $\varepsilon = 1, 0.3$ in section 2.9.1, see fig. 2.13, the improvement in the time-to-solution makes the HOS-SEMFEM approach comparable to the pMG-based preconditioners. In addition, the highly deformed case $\varepsilon = 0.05$ demands the use of SEMFEM as a preconditioner; the relatively poor convergence rate of the pMG approaches yields a much larger time-to-solution. However, with a moderate amount of smoothing, the solve times drop from 2.41 to 2 seconds, a 20% speedup. This latter result makes the HOS-SEMFEM

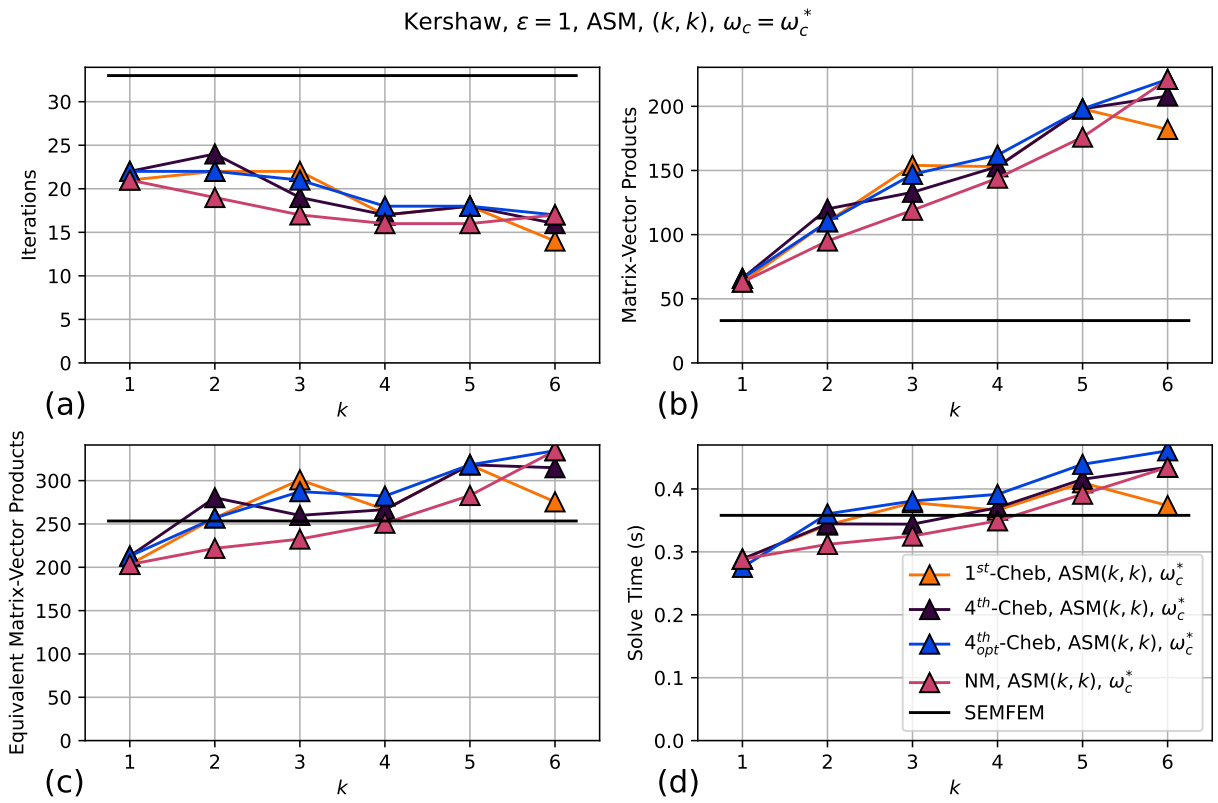



Figure 7.12: Kershaw case from fig. 2.10, $\varepsilon = 1$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k, k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

approach especially attractive for highly deformed problems as the optimal SEMFEM preconditioner is now improved through the smoothing in alg. 6.2.

How does our work metric of equivalent matrix-vector products (eq. (6.13)) shown in subplot (c) of each figure correspond to the actual time-to-solution in subplot(d)? Since the case considered here is on a single node, the communication is entirely *on-node*. As the communication overhead required in the AMG solver is relatively low, corresponding to the left-most data point in fig. 3.19, the time spent in the AMG solver is focused on the matrix-vector products associated with the AMG hierarchy, which is accounted for by $C_{AMG}$ in eq. (6.13). Later in section 7.2.2, we will observe cases where the communication overhead is significant in comparison, especially for the 1568 pebble case.

Figures 7.12 to 7.14, moreover, leads us to conclude that only a relatively *light* amount of smoothing is needed for the scheme proposed in alg. 6.2 to be effective. In particular,

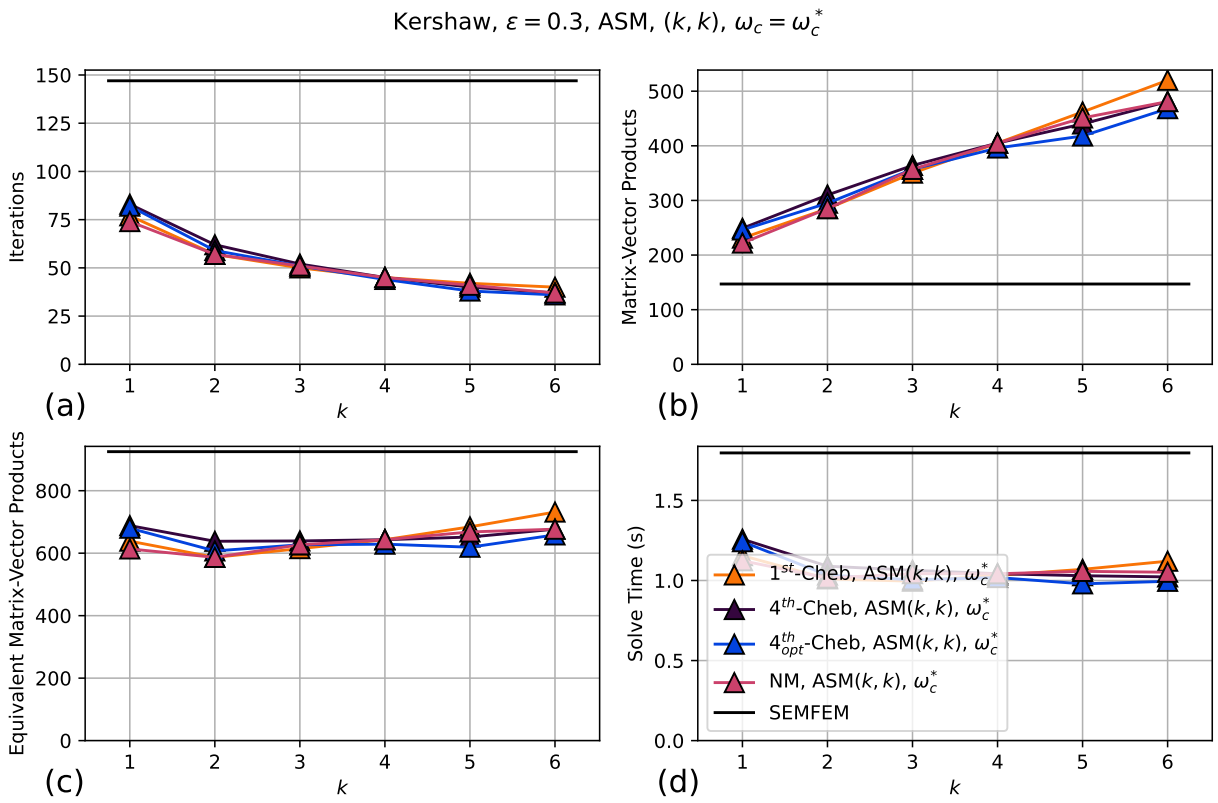



Figure 7.13: Kershaw case from fig. 2.10, $\varepsilon = 0.3$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k,k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

our work metric of equivalent matrix-vector products (eq. (6.13)) shown in subplot (c) of each figure reaches a minimum at $k = 1$ or $k = 2$ for all three Kershaw cases, irrespective of the choice of polynomial smoother. This is in direct contrast to the $p$-multigrid results from section 4.4, wherein the Kershaw $\varepsilon = 0.3$ and $\varepsilon = 0.05$ cases benefited from relatively high smoother polynomial orders with RAS or ASM smoothing. This result, however, is not surprising. The spectra of the high-order operator is roughly equivalent to low-order operator, as shown in the left sub-figure in fig. 2.3. The remaining content spectral content in $A_F^{-1}A$, moreover, is relatively low in magnitude and is concentrated at the highest frequencies.

The observation that a relatively light smoother is needed for the HOS-SEMFEM scheme motivates the authors to consider switch the use of ASM smoothing to Jacobi smoothing. Results similar to those in figs. 7.12 to 7.14 are shown in figs. 7.15 to 7.17. The latter results, however, utilize Jacobi smoothing. For $\varepsilon = 1$, the HOS-SEMFEM approach with

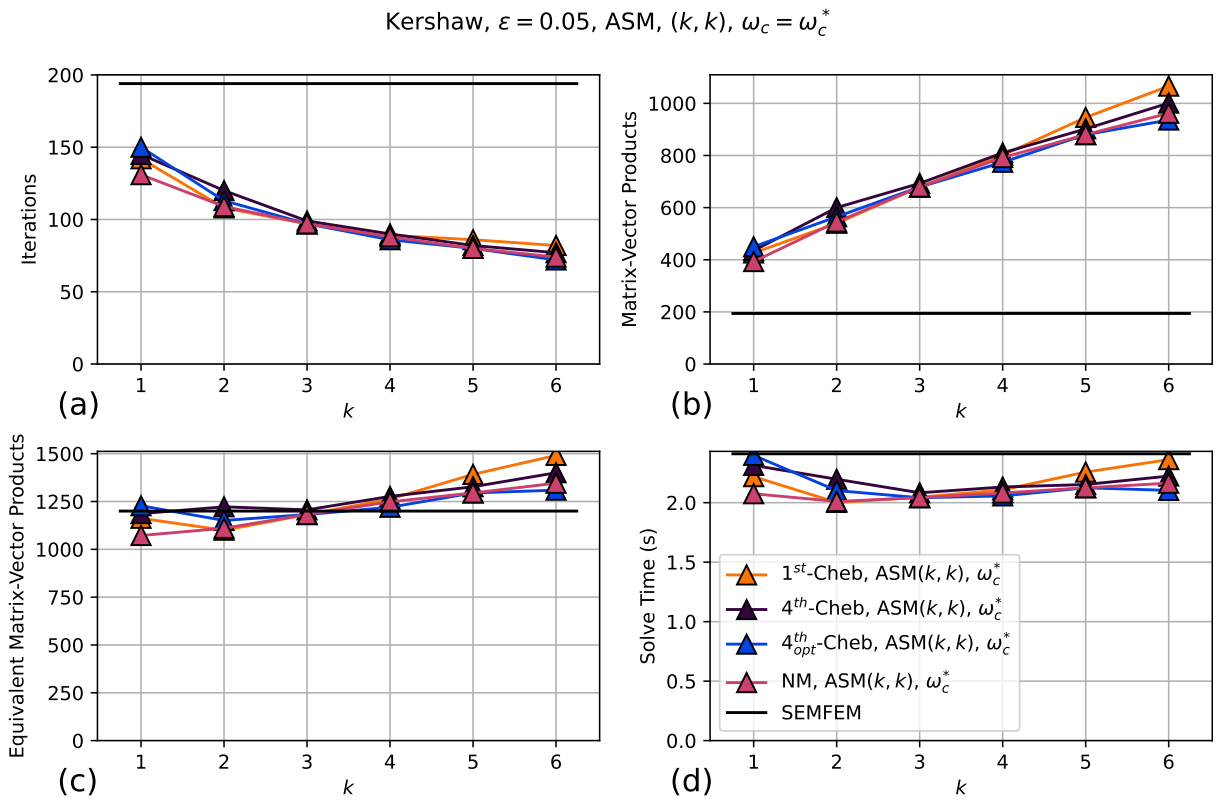



Figure 7.14: Kershaw case from fig. 2.10, $\varepsilon = 0.05$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k,k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

Jacobi yields a negligible reduction in the time-to-solution compared to the same approach with ASM smoothing. The former achieves a time-to-solution of 0.25 seconds, while the latter achieves 0.27 seconds. Similarly for $\varepsilon = 0.05$, the 1.9 second time-to-solution with Jacobi smoothing is only slightly better than the 2.0 second time-to-solution with ASM smoothing. At the same time, the time-to-solution with Jacobi smoothing degrades for $\varepsilon = 0.3$. The HOS-SEMFEM approach with ASM smoothing achieves sub-second time-to-solutions, while the same approach with Jacobi smoothing requires 1.2 seconds. While Jacobi did not improve the time-to-solution for the HOS-SEMFEM method, it remains in striking distance of the ASM results. This is in contrast to the $p$-multigrid results from sections 2.9 and 4.4, wherein Jacobi-based smoothing methods significantly lag behind the ASM- and RAS-based Schwarz smoothers.

An additional consideration in the design of the HOS-SEMFEM scheme is the choice of the

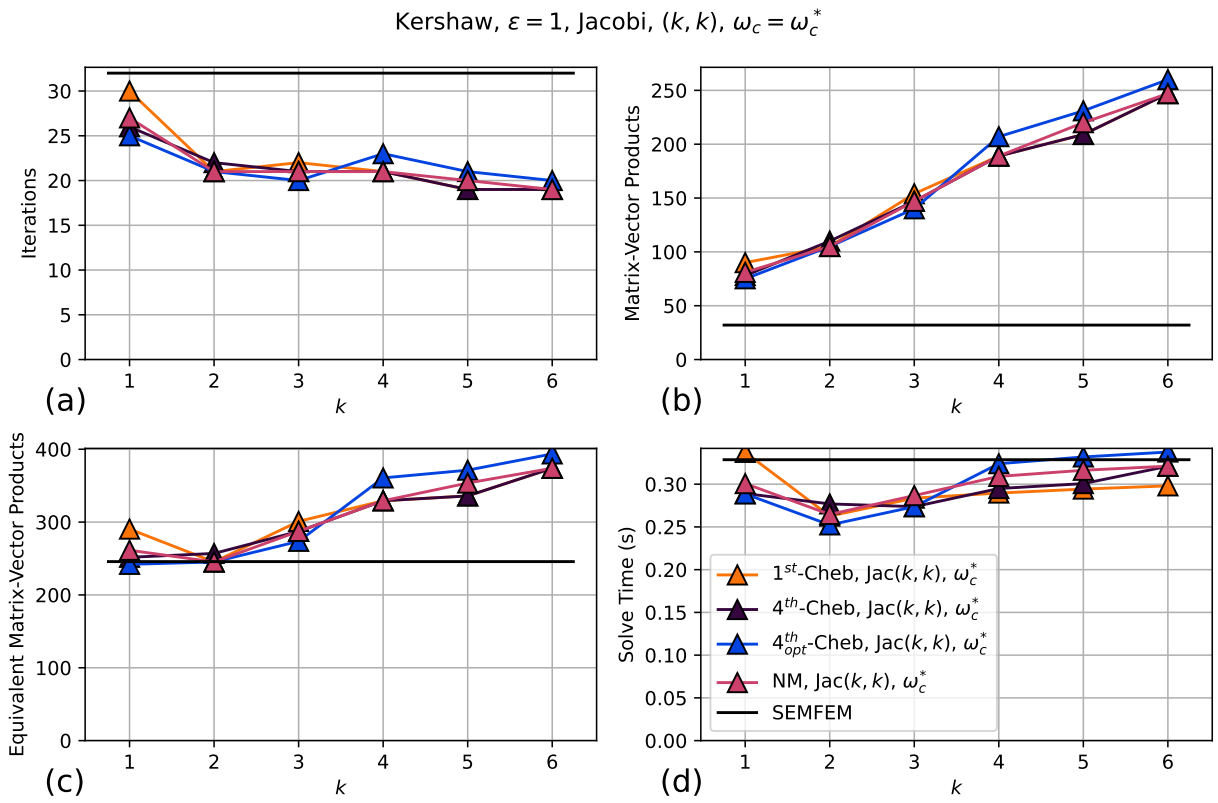



Figure 7.15: Kershaw case from fig. 2.10, $\varepsilon = 1$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k,k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

polynomial smoother. While the polynomials directly optimized via Nelder-Mead from section 4.5 are expected to outperform the other smoother polynomials, the results in figs. 7.12 to 7.14 and figs. 7.15 to 7.17 suggest that the choice of polynomial smoother is not critical. The choice of smoothing polynomial in alg. 6.2 has a relatively small impact on the overall solver performance. This is in stark contrast to the $p$-multigrid results from section 4.4, wherein solver performance quickly degraded for the $1^{st}$-Cheb smoother polynomials with respect to higher smoothing orders. However, the coarse-grid space considered in the HOS-SEMFEM approach is much *richer* than the coarse-grid space considered in the $p$-multigrid approach or for which the $4^{th}$-Cheb and $4^{th}_{opt}$-Cheb smoother polynomials are constructed in section 4.1 Further, as the NM polynomials from section 4.5 use the fourth-kind Chebyshev polynomials as an initial condition for the optimizer, the optimizer may converge to a local minima near the initial condition. In this case, the resulting NM polynomials would be similar to the $4^{th}$-Cheb smoother polynomials.

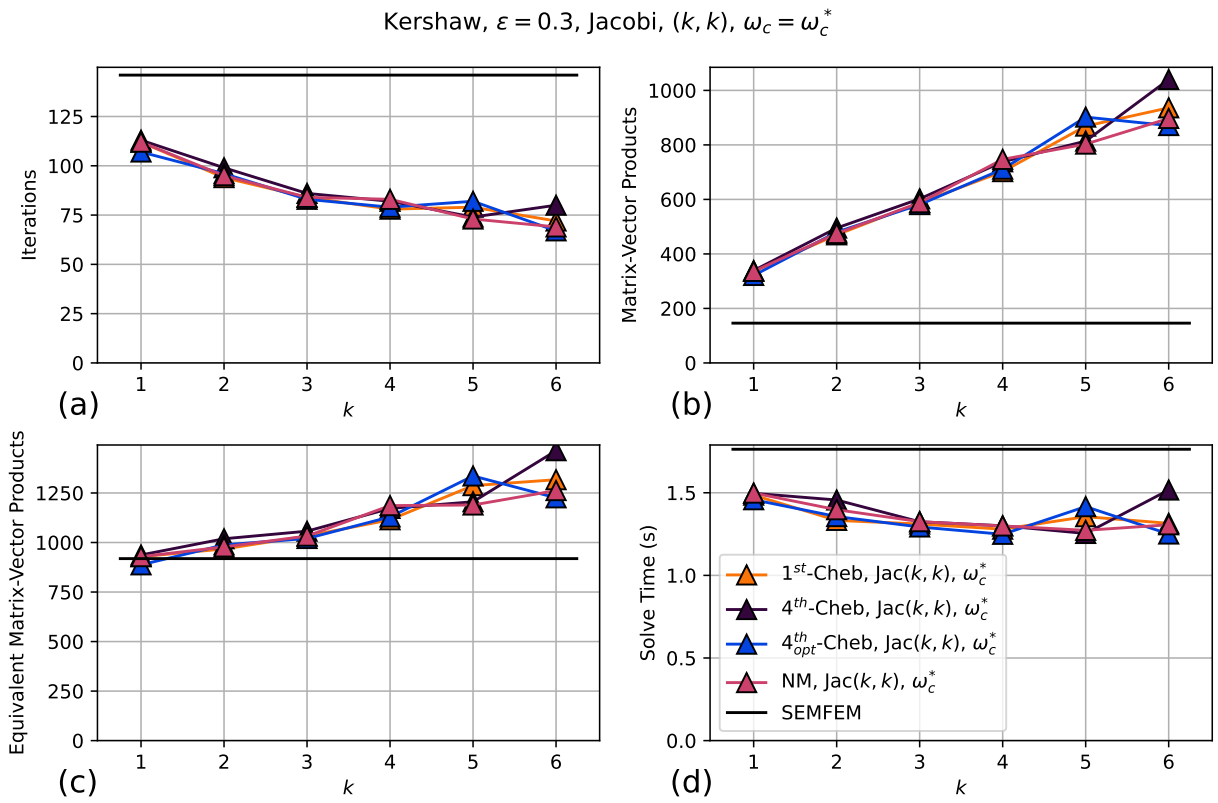



Figure 7.16: Kershaw case from fig. 2.10, $\varepsilon = 0.3$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k,k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

Additional results, including omitting the post-smoother and considering $\omega_c = 1$, are shown in appendix B.2.1. The choice of $(k,k)$ versus $(2k,0)$ smoothing does not make a significant impact on the solver performance. Further, tuning $\omega_c$ improves the solver performance, but choosing $\omega_c = 1$ still results in an improvement using the HOS-SEMFEM approach when compared to SEMFEM.

### 7.2.2 Pebble Cases

In section 2.8.2, we introduced three pebble cases with 146, 1568, and 67 spheres. These cases ([53, 54, 55]) are real complex geometries of interest that are time-varying Navier-Stokes calculations. However, to facilitate the comparison against many different solver parameters without requiring the full Navier-Stokes simulation, the right-hand side for the resultant Pressure Poisson problem from the final solution step of the Navier-Stokes solver is used

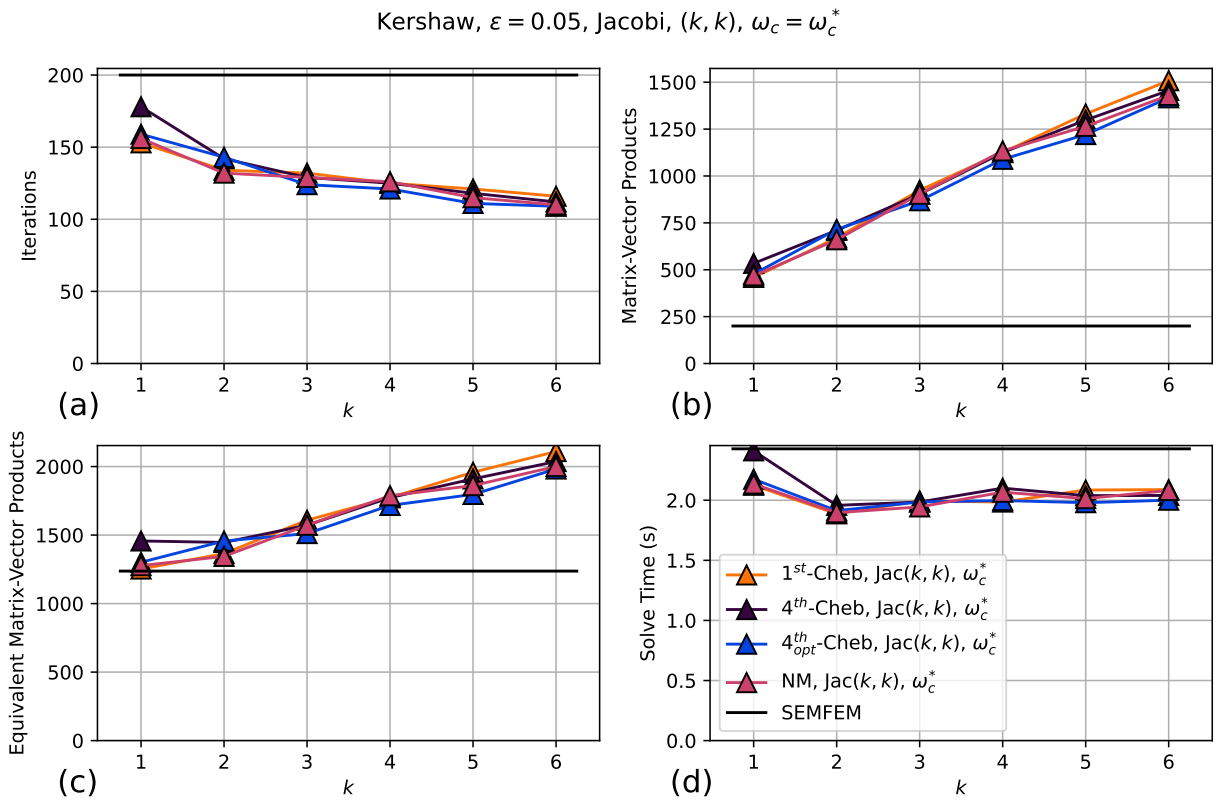



Figure 7.17: Kershaw case from fig. 2.10, $\varepsilon = 0.05$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k,k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

as the test problem. For tuning $\omega_c$ and the NM smoother polynomials from section 4.5, a random right-hand side is constructed by taking a random solution vector on $[0,1]$ satisfying the boundary condition and applying the discrete Laplacian operator.

Results using the HOS-SEMFEM solver outlined in alg. 6.2 are shown in figs. 7.18 to 7.20. Here, the ASM smoother from alg. 2.10 is accelerated using a polynomial smoother with smoother passes distributed symmetrically in the pre- and post-smoothing phases of the "two-level" cycle described in alg. 6.2. This smoother is denoted as $(k,k)$ from chapter 4. Similar results utilizing Jacobi smoothing are also shown in figs. 7.21, 7.24 and 7.25.

For the 146 pebble case (fig. 7.18), we observe a large reduction in the iteration count from 34 to 20 using relatively light $1^{st}$-Cheb, ASM(1,1) smoothing. This has the effect of reducing the time-to-solution from 0.45 to 0.34 seconds, a 32% speedup. Similar to the results in section 7.2.1, the equivalent matrix-vector product metric (eq. (6.13)) shown in fig. 7.18(c) accurately predicts the relative performance of the solvers. This is expected,

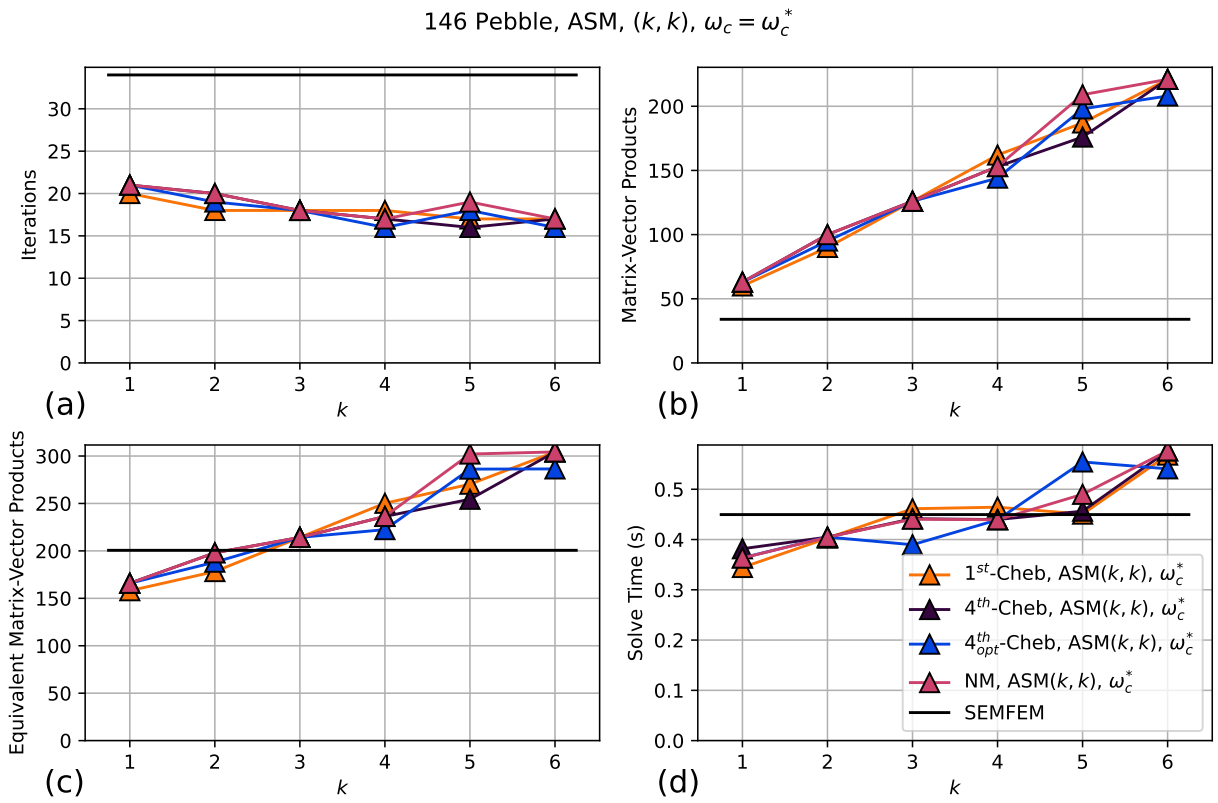



Figure 7.18: 146 pebble case from fig. 2.11a. One node of Summit ($P = 6$ V100 GPUs) is used, $n/P \sim 3.5M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k,k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

as the 146 pebble case runs on a single node of Summit. Therefore, the communication overhead associated with the AMG solves are minimal. Relatively light-weight smoothing yields the lowest time-to-solution; the minimal time to solution is reached with $k = 1$ using $1^{st}$-Cheb, ASM(1,1) smoothing.

How does ASM smoothing compare to Jacobi smoothing? Comparing fig. 7.21 to fig. 7.18, we observe that $4^{th}_{opt}$-Cheb, Jacobi(1,1) smoothing yields a slightly higher iteration count of 20, compared to 20 for $1^{st}$-Cheb, ASM(1,1). However, the time-to-solution is slightly lower for $4^{th}_{opt}$-Cheb, Jacobi(1,1) smoothing at 0.32 seconds, compared to 0.34 seconds for $1^{st}$-Cheb, ASM(1,1) smoothing. This improvement, however, is not significant.

While ASM smoothing proved effective for the HOS-SEMFEM preconditioner in the 146 pebble case, the same cannot be said for the 67 pebble case (fig. 7.19). At $k = 1$, the iteration count is reduced from 38 to 35 iterations with $4^{th}_{opt}$-Cheb, ASM(1,1) smoothing. However, this translates to roughly the same time-to-solution—0.5 seconds for SEMFEM and 0.48

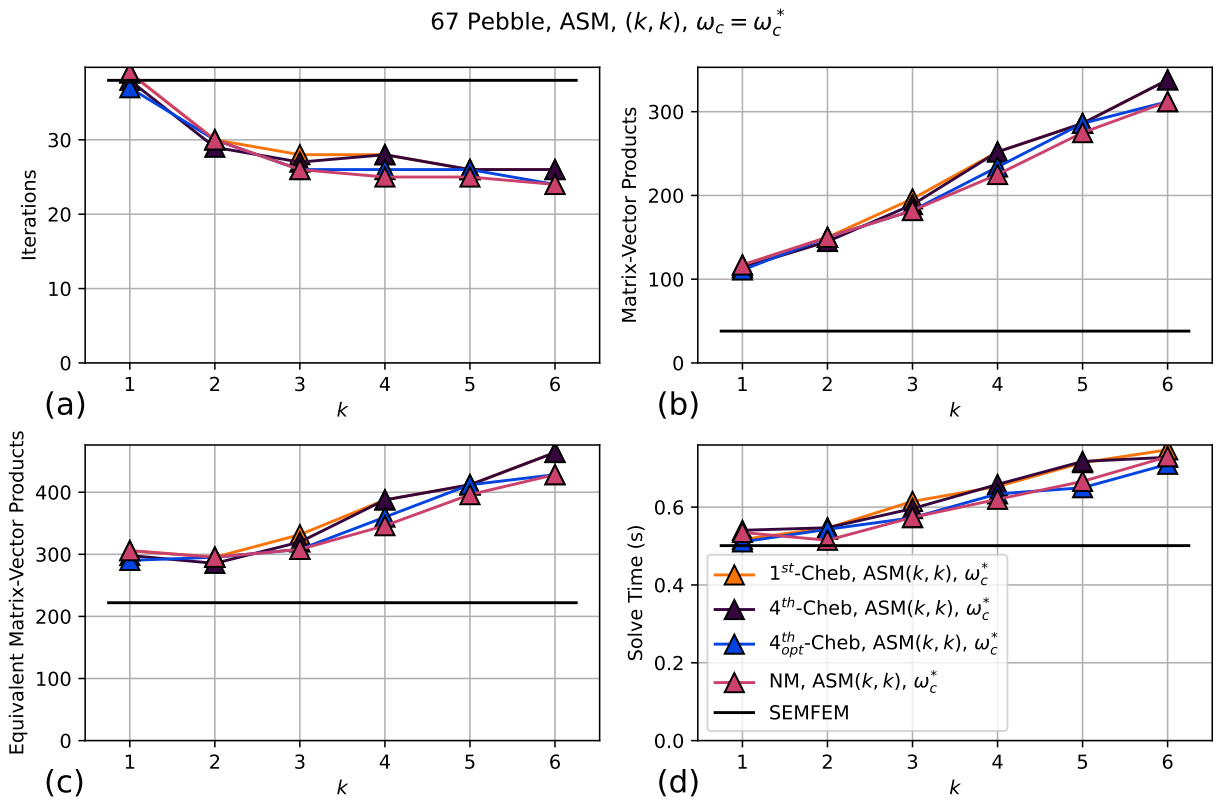



Figure 7.19: 67 pebble case from fig. 2.11c. Three nodes of Summit ($P = 18$ V100 GPUs) is used, $n/P \sim 2.33M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k,k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

seconds using the HOS-SEMFEM preconditioner. Jacobi smoothing (fig. 7.24), however, proves much more effective. NM, Jac(1,1) smoothing reduces the iteration count to 27. This large iteration count reduction, combined with the use of an inexpensive smoother, translates to a time-to-solution of 0.36 seconds, a 38% speedup over the SEMFEM solver.

Why does ASM smoothing prove ineffective for the 67 pebble case? The ASM smoother from alg. 2.10 is significantly more expensive than simple diagonal smoothing, costing roughly 3 times as much as a matrix-vector product (see, for example, the results in fig. 3.19). Usually, this expense is justified by the smoothing properties of ASM smoothing. However, as observed in the paragraph, ASM smoothing yields *larger* iteration counts compared to Jacobi smoothing. As mentioned in the previous paragraph, the use of ASM smoothing is worse *both* in terms of iteration count and time-to-solution. The sharp, connectivity of the 67 pebble case as depicted in fig. 2.11c is a contributing factor in the poor performance of ASM smoothing. For example, 11 spectral elements share a single vertex in at least one of

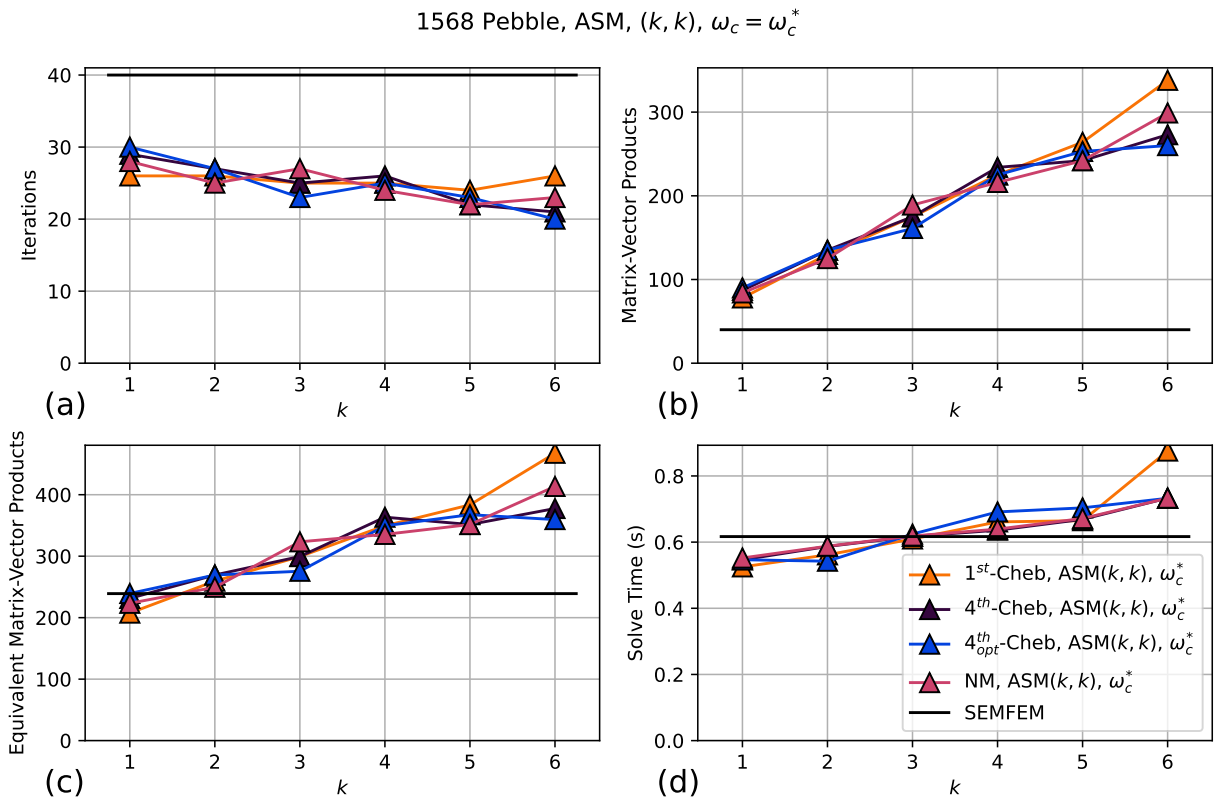



Figure 7.20: 1568 pebble case from fig. 2.11b. 12 nodes of Summit ($P = 72$ V100 GPUs) is used, $n/P \sim 2.5M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k,k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

the pebbles. While this geometric irregularity is well-resolved by the low-order operator, the ASM smoother, which is constructed using box-like approximations of each spectral element, does not improve the convergence rate of the solver. This proves to be a pathological case for ASM smoothing. Jacobi smoothing, on the other hand, improves the solution quality at the pebble vertices, improving the convergence rate of the HOS-SEMFEM scheme.

To demonstrate this effect, we visualize the residual in each step of the HOS-SEMFEM scheme for the 67 pebble case for ASM smoothing in fig. 7.22 and Jacobi smoothing in fig. 7.23. The relatively large residual in the top-left pebble near the chamfer remains largely *undamped* by the ASM smoother moving from fig. 7.22a to fig. 7.22b. After applying the SEMFEM coarse-grid correction and another pass of ASM smoothing, we observe that the residual in the top-left pebble remains large in fig. 7.22d. In contrast, the residual in the top-left pebble is damped by the Jacobi smoother in fig. 7.23b. Here, the choice of Jacobi smoothing results in a final residual that is smaller than the ASM smoother in fig. 7.23d.

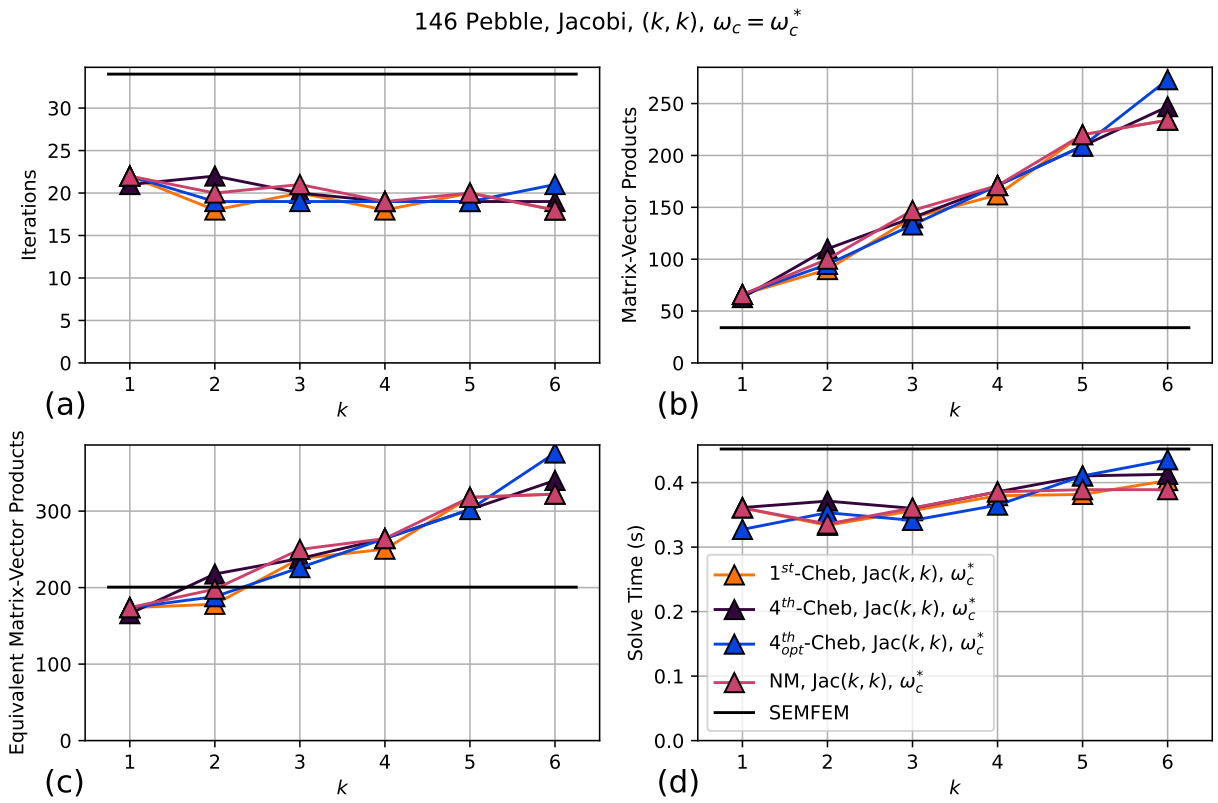
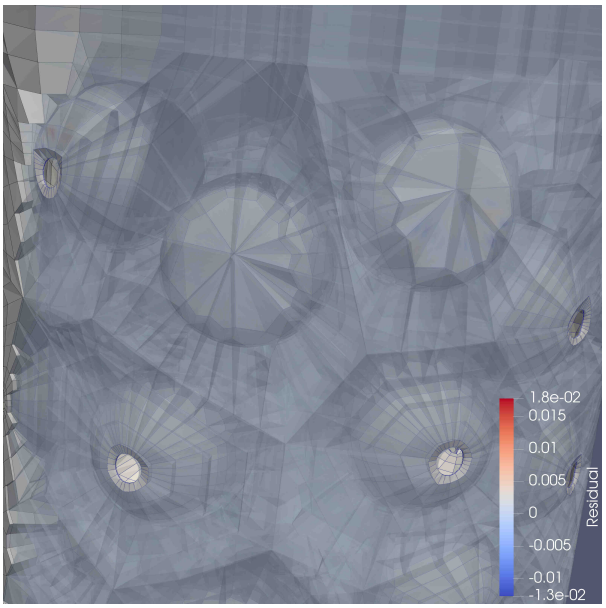



Figure 7.21: 146 pebble case from fig. 2.11a. One node of Summit ($P = 6$ V100 GPUs) is used, $n/P \sim 3.5M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k, k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

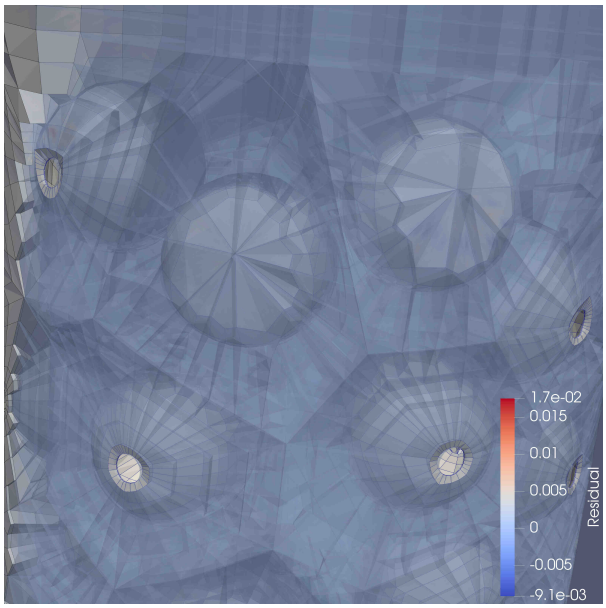
Remarkably, a *cheaper* smoother yields a *better* solution. For the other cases, however, we observe similar iteration counts amongst the HOS-SEMFEM schemes with Jacobi and ASM smoothing. All else being equal, we recommend the use of Jacobi smoothing here for its lower cost.

While the 67 pebble case only encompasses three nodes of Summit ($P = 18$ V100 GPUs), the effect of *off-node* communication costs is sufficiently large that our equivalent matrix-vector product metric in fig. 7.24c can no longer serve as a proxy for the time-to-solution. While the equivalent matrix-vector product metric climbs with respect to the smoother polynomial order, we observe that the time-to-solution remains relatively constant. This effect becomes more pronounced in the 1568 pebble case.

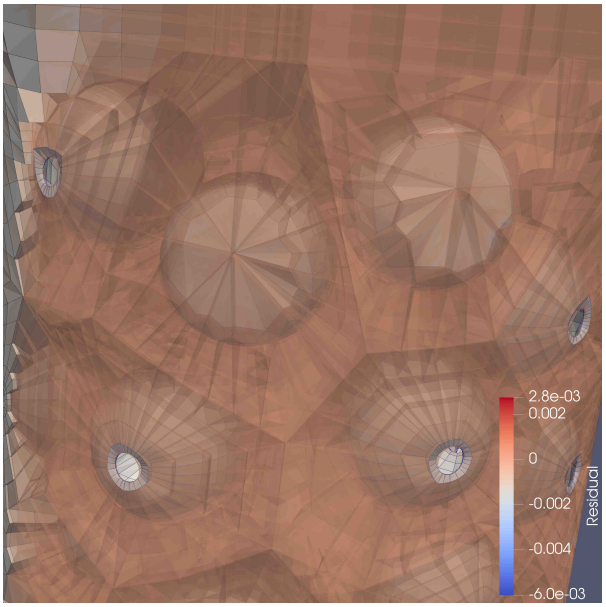
In the 1568 pebble case (figs. 7.20 and 7.25), the iteration count is improved from 40 to 27 iterations using $1^{st}$-Cheb, ASM(1,1) smoothing. This translates to an improvement in the time-to-solution from 0.62 to 0.51 seconds using the HOS-SEMFEM preconditioner, a 22%



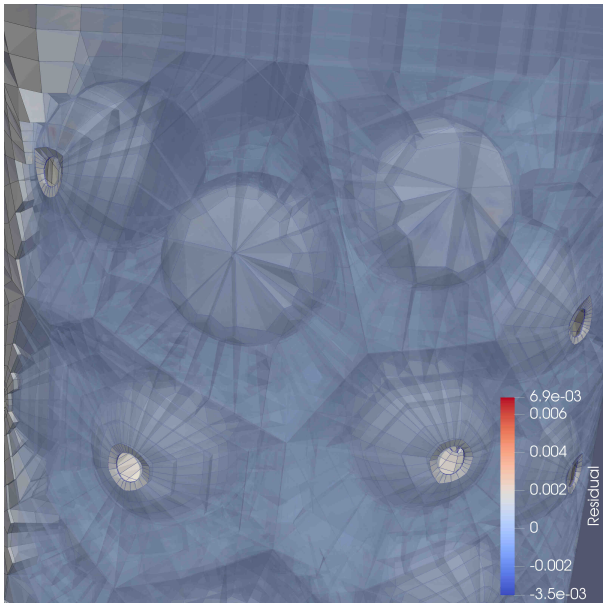

(a) Initial residual $\underline{r}$ in preconditioner $\underline{z} = M^{-1}\underline{r}$ computation.




(b) Residual after smoothing, prior to SEMFEM correction.



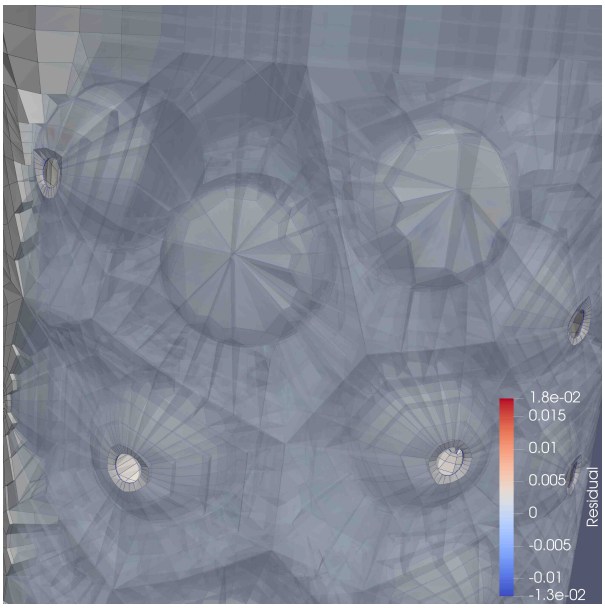

(c) Residual after SEMFEM correction.



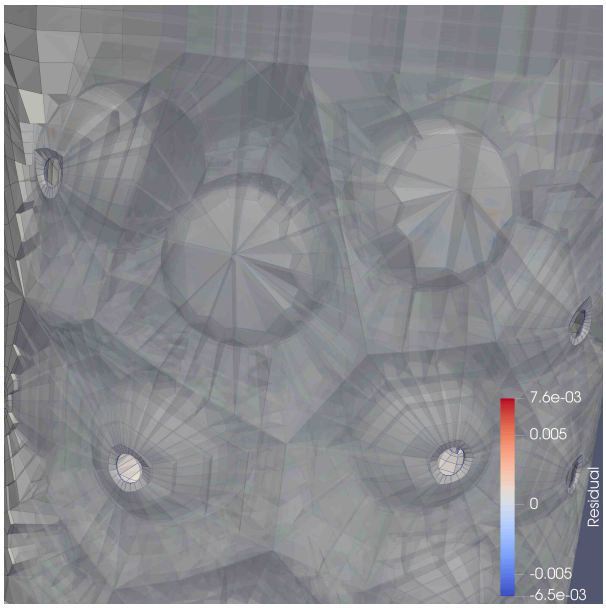

(d) Final residual at end of preconditioner.

Figure 7.22: Residual visualization for the 67 pebble case in (fig. 2.11c) using HOS-SEMFEM scheme with optimal coarse-grid relaxation $\omega_c = \omega_c^*$ parameter and $4^{th}_{opt}$-Cheb, ASM(1,1) smoothing.

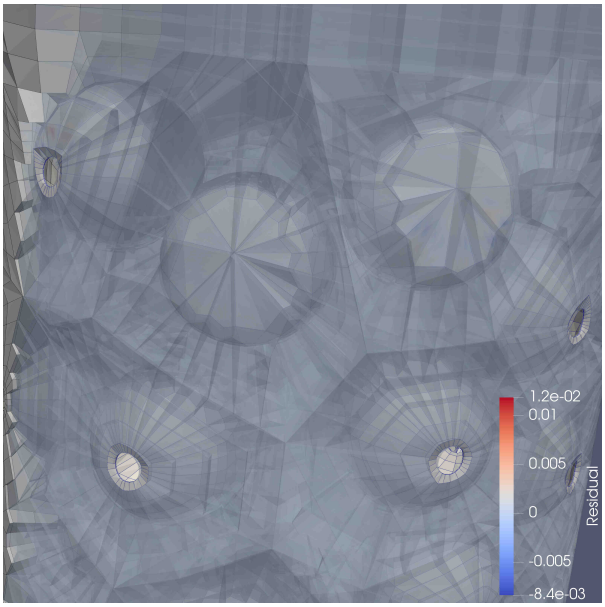
speedup. However, the time-to-solution is slightly lower for $1^{st}$-Cheb, Jacobi(2,2) smoothing, which achieves a time-to-solution of 0.44 seconds. This represents a speedup of 41% utilizing the HOS-SEMFEM approach compared to SEMFEM.



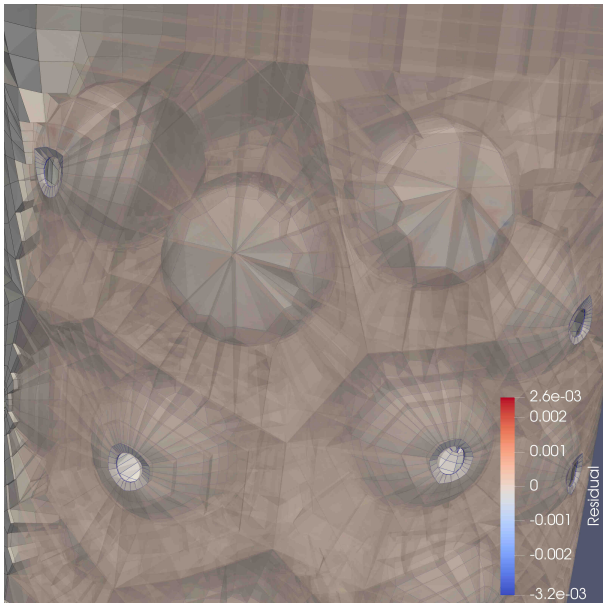

(a) Initial residual $\underline{r}$ in preconditioner $\underline{z} = M^{-1}\underline{r}$ computation.




(b) Residual after smoothing, prior to SEMFEM correction.




(c) Residual after SEMFEM correction.




(d) Final residual at end of preconditioner.

Figure 7.23: Residual visualization for the 67 pebble case in (fig. 2.11c) using HOS-SEMFEM scheme with optimal coarse-grid relaxation $\omega_c = \omega_c^*$ parameter and $4^{th}_{opt}$-Cheb, Jacobi(1,1) smoothing.

At the modest scale of 12 nodes of Summit ($P = 72$ V100 GPUs), we observe significant off-node communication costs. For example, fig. 7.25c shows that the equivalent matrix-vector product count is roughly equivalent at 221 work-units for the HOS-SEMFEM scheme

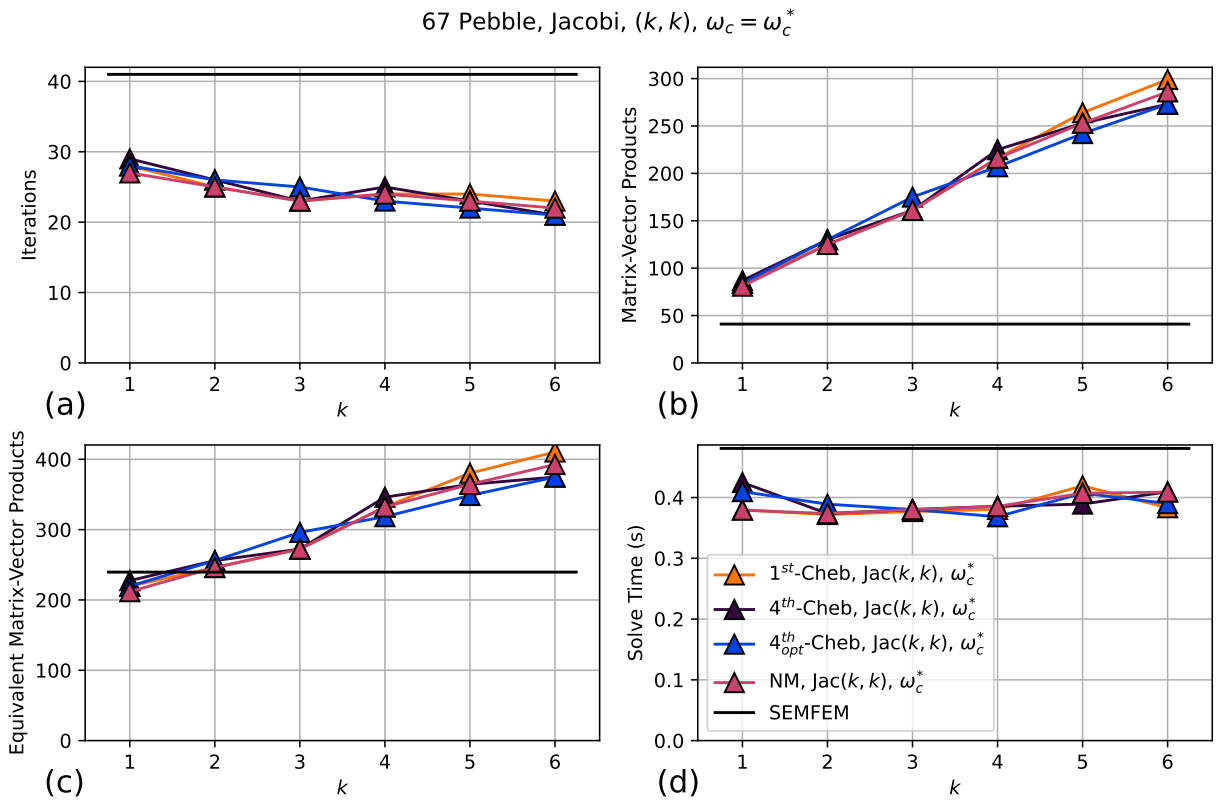



Figure 7.24: 67 pebble case from fig. 2.11c. Three nodes of Summit ($P = 18$ V100 GPUs) is used, $n/P \sim 2.33M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k,k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

and the SEMFEM preconditioner at $k = 1$. As the smoother polynomial order is increased, however, this metric increases by over a factor of 2 at $k = 6$. During this time, however, the time-to-solution remains relatively unchanged. The minimum time-to-solution, moreover, is achieved at $k = 2$ with the $1^{st}$-Cheb, Jacobi(2,2) smoothing strategy, which corresponds to approximately 269 work-units.

In order to understand the effect of tuning the value of $\omega_c$, results using $\omega_c = 1$ are presented in appendix B.2.2. Tuning the value of $\omega_c$ improves performance, but choosing $\omega_c = 1$ still results in a solver that is able to improve the time-to-solution compared to SEMFEM. Additional results exploring the $(2k, 0)$ V-cycle strategy are also presented in appendix B.2.2.

The results presented in sections 7.2.1 and 7.2.2 are summarized in table 7.2. The HOS-SEMFEM approach is always able to improve on the time-to-solution when compared to the SEMFEM solver, assuming that the smoother is chosen correctly. Remarkably, Jacobi-

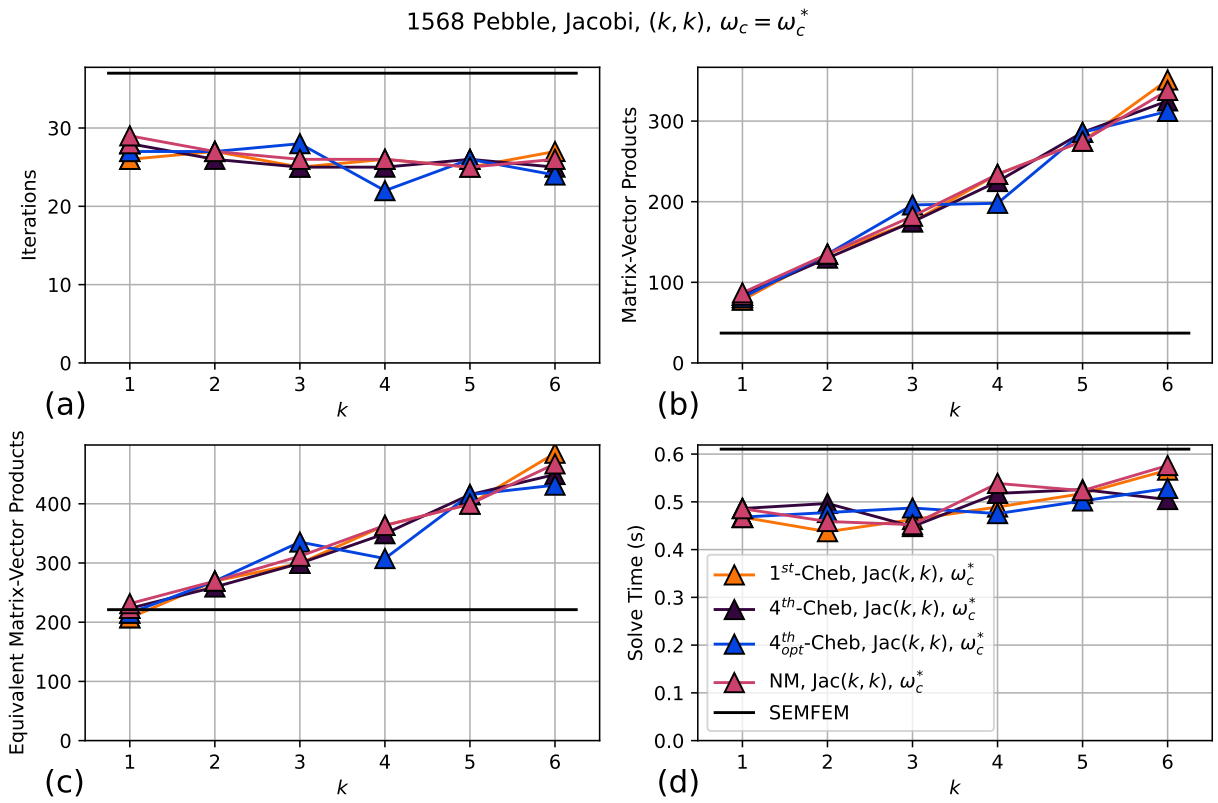



Figure 7.25: 1568 pebble case from fig. 2.11b. 12 nodes of Summit ($P = 72$ V100 GPUs) is used, $n/P \sim 2.5M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The symmetric distribution of pre- and post-smoothing passes, denoted by $(k,k)$ from chapter 4, is used. The polynomial smoother order, $k$, is varied.

based smoothing approaches tend to outperform ASM smoothing, albeit by a small amount in most cases. For the 67 pebble case, however, Jacobi smoothing must be used to reach a significant speedup in the time-to-solution. Tuning the value of $\omega_c$ is required to reach the optimal solvers, with exception to the 67 pebble case. In this scenario, $4^{th}$-Cheb, Jacobi(2,2) smoothing with $\omega_c = 1$ outperforms the tuned $\omega_c^*$ value. This improvement, however, is within the noise on Summit. The optimized smoother polynomials from section 4.5 and tuned $\omega_c^*$ NM, Jac(1,1) solver is able to achieve a time-to-solution of 0.36 seconds.

The results from table 7.2 are important for two reasons. First, SEMFEM was previously the best solver for the 1568 and 67 pebble cases. Moreover in the latter case, SEMFEM achieved nearly a factor 4 speedup in the time-to-solution from section 2.9. However, given the correct choice of smoother, polynomial smoother, and smoothing order, the HOS-SEMFEM approach improves on the SEMFEM time-to-solution by 33% and 38% for the 67 and 1568 pebble cases, respectively. Second, the HOS-SEMFEM approach *extends* the

Table 7.2: Solver configuration yielding the lowest time-to-solution in seconds, $T_s$, for the HOS-SEMFEM approach from chapter 6. The time-to-solution for the SEMFEM solver, $T_{S-F}$, speedup $T_{S-F}/T_s$, iteration counts for the solver, $I_s$, and iteration count for SEMFEM, $I_{S-F}$, are shown.

| Case | Preconditioner | $T_s$ | $T_{S-F}$ | $\frac{T_{S-F}}{T_s}$ | $I_s$ | $I_{S-F}$ |
|---|---|---|---|---|---|---|
| Kershaw, $\varepsilon = 1$ | $4^{th}_{opt}$-Cheb, Jacobi(2,2), $\omega_c^*$ | 0.25 | 0.33 | 1.32 | 21 | 32 |
| Kershaw, $\varepsilon = 0.3$ | $4^{th}$-Cheb, ASM(8,0), $\omega_c^*$ | 0.97 | 1.80 | 1.86 | 42 | 147 |
| Kershaw, $\varepsilon = 0.05$ | $1^{st}$-Cheb, Jacobi(2,2), $\omega_c^*$ | 1.89 | 2.43 | 1.29 | 134 | 200 |
| 146 pebble | $4^{th}_{opt}$-Cheb, Jacobi(1,1), $\omega_c^*$ | 0.33 | 0.45 | 1.36 | 22 | 34 |
| 67 pebble | $4^{th}$-Cheb, Jacobi(2,2), $\omega_c = 1$ | 0.36 | 0.48 | 1.33 | 27 | 41 |
| 1568 pebble | $1^{st}$-Cheb, Jacobi(2,2), $\omega_c^*$ | 0.44 | 0.61 | 1.38 | 27 | 37 |

region in which the low-order refined preconditioning strategies make sense. By improving the time-to-solution drastically in the Kershaw $\varepsilon = 0.3$ case, the HOS-SEMFEM approach is now within striking distance of the pMG-based methods, which, as shown in fig. 4.12, outperform the low-order preconditioners.

## 7.3 COMPARING THE BEST SOLVERS

In section 7.1, we consider the performance of the additive coarse-grid solve proposed in chapter 5 to the standard multiplicative method and determined the best solver parameters for each case, given the choice of preconditioner. These results are in table 7.1. Section 7.2, on the other hand, consider the performance of a HOS-SEMFEM approach, which is an auxiliary space method utilizing the low-order operator as a coarse-space, from chapter 6. We compare the efficacy of this method to the standard low-order preconditioner from section 2.4. The optimal solver configuration for the HOS-SEMFEM solver is given in table 7.2.

In this section, we now compare the best methods from sections 7.1 and 7.2 to determine the best solver for each case. Taking the solver with the lowest time-to-solution from tables 7.1 and 7.2, the best solver for each case is given in table 7.3.

As we observe in table 7.3, the optimal solver configuration still varies on a case-by-case basis. *There is no universally optimal solver configuration.* However, we observe a few important trends from these results. First, when combined with the polynomial smoothers from chapter 4 and the ASM smoothers from section 2.6, geometric $p$-multigrid approaches tend to be the most effective preconditioners across a wide-range of problems. As mentioned in section 3.3, however, the coarse-grid solve required at the $p = 1$ level poses a

Table 7.3: Solver configuration yielding the lowest time-to-solution in seconds, $T_s$.

| Case | Solver | $T_s$ | Iterations |
|---|---|---|---|
| Kershaw, $\varepsilon = 1$ | NM, ASM(1,1),(7,3,1) | 0.08 | 9 |
| Kershaw, $\varepsilon = 0.3$ | NM, ASM(6,0),(7,3,1) | 0.65 | 43 |
| Kershaw, $\varepsilon = 0.05$ | HOS-SEMFEM, $1^{st}$-Cheb, Jacobi(2,2), $\omega_c^*$ | 1.89 | 134 |
| 146 pebble | NM, ASM(3,3),(7,3,1) | 0.27 | 12 |
| 67 pebble | HOS-SEMFEM, $4^{th}$-Cheb, Jacobi(2,2), $\omega_c = 1$ | 0.36 | 41 |
| 1568 pebble | NM, ASM(6,0),(7,3,1) | 0.39 | 18 |

significant bottleneck for the parallel scalability of the method, especially on heterogeneous architectures. However, this issue is readily mitigated by constructing *robust* smoothers, such as ASM combined with the polynomial smoothers from chapter 4. By using a heavier smoother, fewer iterations are required to reach convergence. This has the effect of limiting the number of coarse-grid solves required, as one is needed *per V-cycle iteration*.

The HOS-SEMFEM approach from chapter 6 is also a viable alternative. This approach is especially effective when the low-order preconditioner is already a viable preconditioner for the problem. As we observe in section 7.1, there exists a solver configuration for the HOS-SEMFEM approach that is able to *outperform* the low-order preconditioner. In particular, for Kershaw with $\varepsilon = 0.05$ and the 67 pebble case, the HOS-SEMFEM approach is able to reduce the time-to-solution of the low-order preconditioner by 29% and 33%, respectively. This is a significant improvement, especially considering that the low-order preconditioner is the best solver without the approach from chapter 6. For the 67 pebble case, for example, the time-to-solution is reduced by as much as a factor of 4 when compared to the $p$-multigrid methods (see section 2.9). Further, we observe that the choice of smoother in the HOS-SEMFEM approach matters. For example, relatively light polynomial Jacobi smoothing tends to be more effective than the ASM smoother. This is not, however, universal. For example, the HOS-SEMFEM preconditioner for the Kershaw case with $\varepsilon = 0.3$ worked best with a high-order polynomial ASM smoother with no post-smoothing, see table 7.2.

### 7.3.1 Scalability of the Best Solvers

To better assess the performance of the best preconditioners identified in sections 7.1 and 7.2, we now consider the Kershaw case with $\varepsilon = 1, 0.3, 0.05$. The number of GPUs utilized, $P$, is varied from a single Summit node ($P = 6$) up to 1024 Summit nodes ($P = 6144$). As this is a weak scaling study, the gridpoints per GPU is fixed to $n/P = 2.67M$ degrees of freedom per GPU.

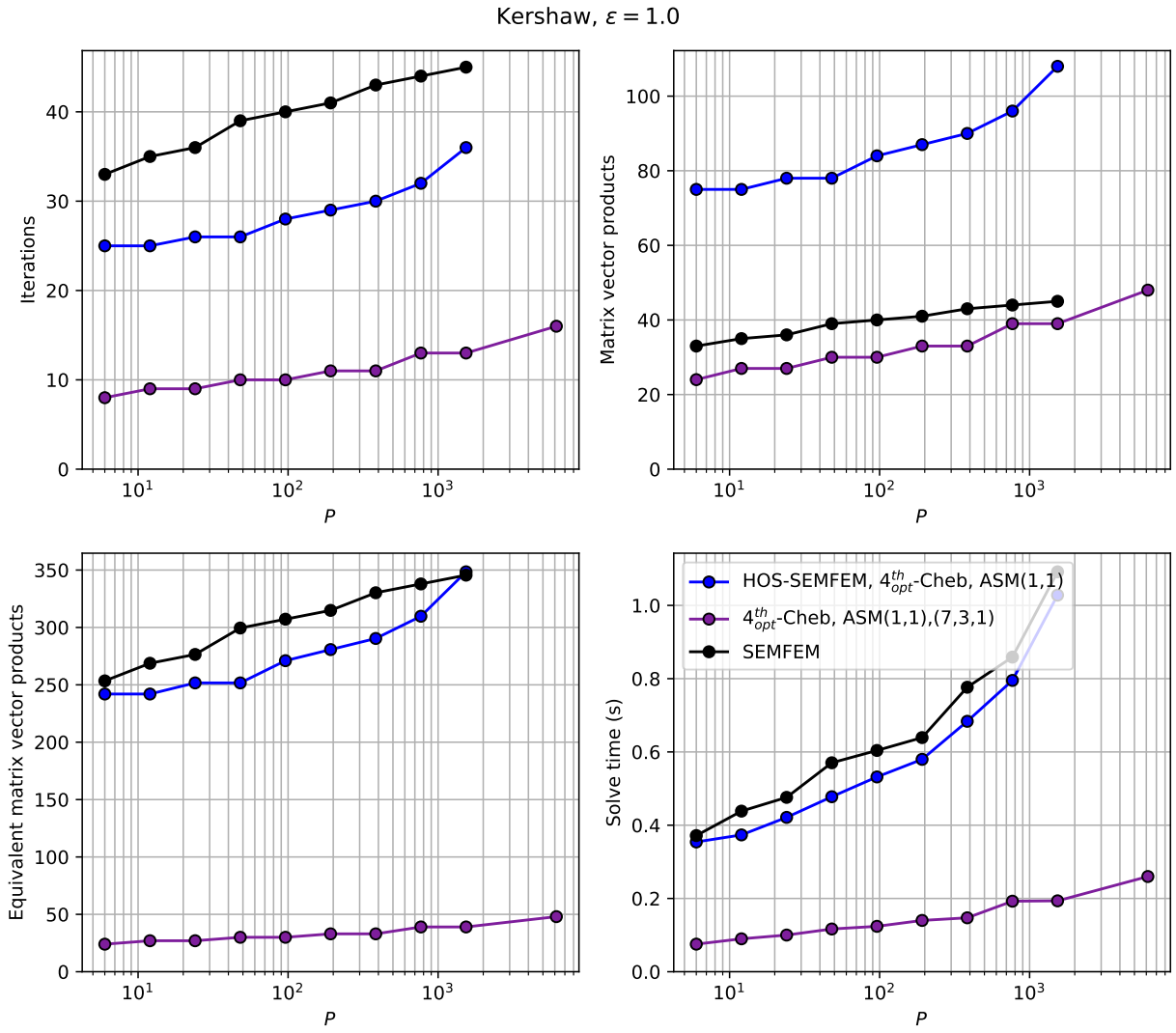



Figure 7.26: Weak scaling results for the Kershaw problem with $\varepsilon = 1.0$. The number of GPUs, $P$, is varied from $P = 6$ to $P = 6144$. The number of gridpoints per GPU is fixed to $n/P = 2.67M$. For a fair comparison, the best solver configuration for each case *at scale* is used. For simplicity, we consider $\omega_c = 1$ for the HOS-SEMFEM scheme.

Figures 7.26 to 7.28 show the weak scaling results for the Kershaw problem with $\varepsilon = 1, 0.3, 0.05$, respectively. To enable work comparison between the solvers, both the matrix-vector product count and *equivalent matrix-vector product count* from eq. (6.13) are shown. The latter metric includes an estimate for the work performed in the AMG V-cycle for the low-order preconditioners.

The best solver configurations identified for the Kershaw problem *on a single node* remain the best solver configurations for the Kershaw problem *at scale*, according to figs. 7.26 to 7.28. For the relatively east $\varepsilon = 1$ case, geometric $p$-multigrid built using a relatively

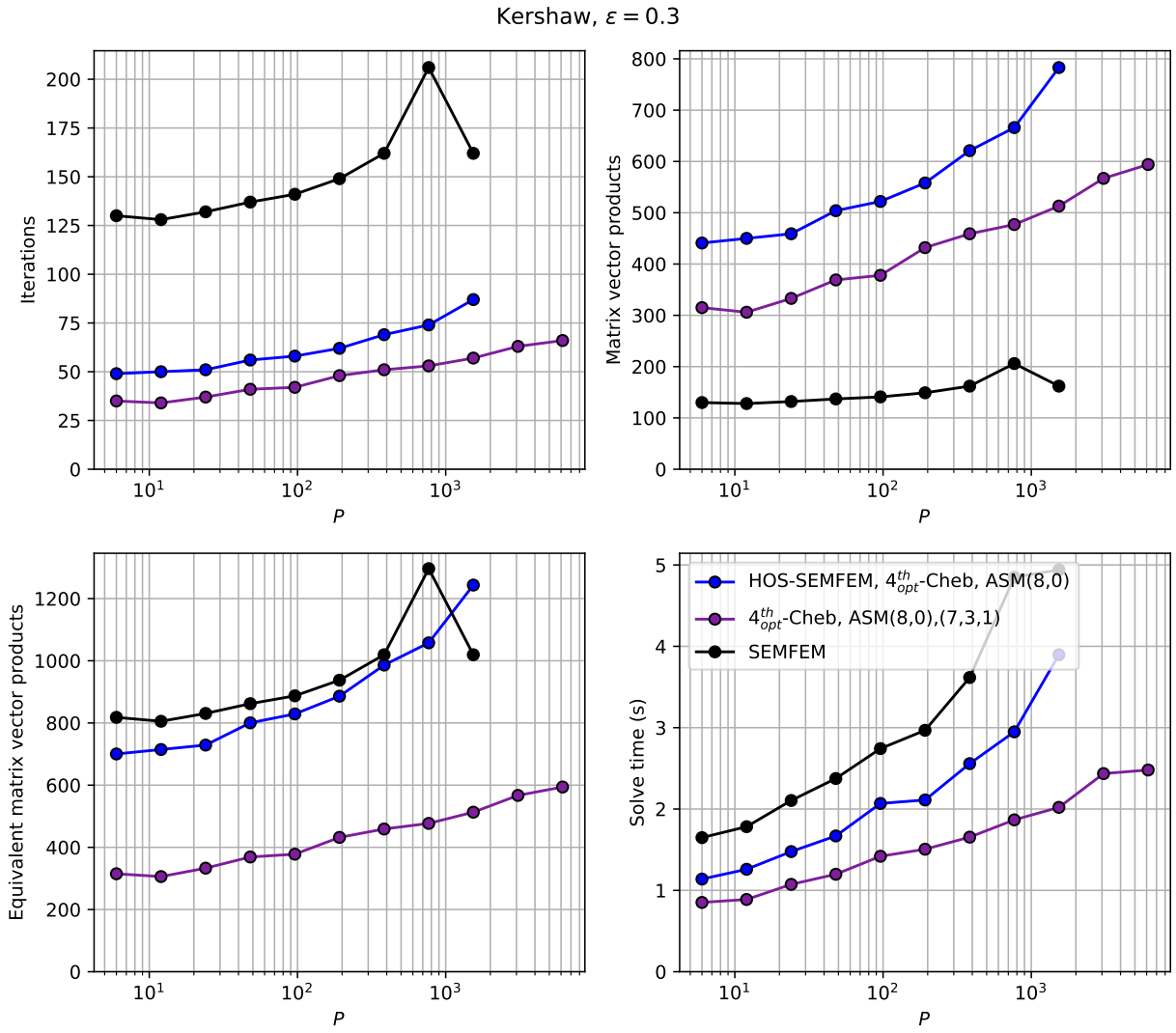



Figure 7.27: Weak scaling results for the Kershaw problem with $\varepsilon = 0.3$. The number of GPUs, $P$, is varied from $P = 6$ to $P = 6144$. The number of gridpoints per GPU is fixed to $n/P = 2.67M$. For a fair comparison, the best solver configuration for each case *at scale* is used. For simplicity, we consider $\omega_c = 1$ for the HOS-SEMFEM scheme.

light polynomial smoother outperform the SEMFEM and HOS-SEMFEM approaches. This remains the case for the moderate mesh deformation case with $\varepsilon = 0.3$. While the *gap* between the HOS-SEMFEM and $p$-multigrid approaches is reduced, the $p$-multigrid approaches remain the best solvers. However, when considering the highly deformed $\varepsilon = 0.05$ case, the HOS-SEMFEM approach is the best solver. Notably, across all cases, the HOS-SEMFEM approach *improves* the time to solution in comparison to the SEMFEM preconditioner.

When the $p$-multigrid preconditioner is viable, it displays the best parallel scalability of the preconditioners considered. However, as in the highly-deformed case with $\varepsilon = 0.05$, the

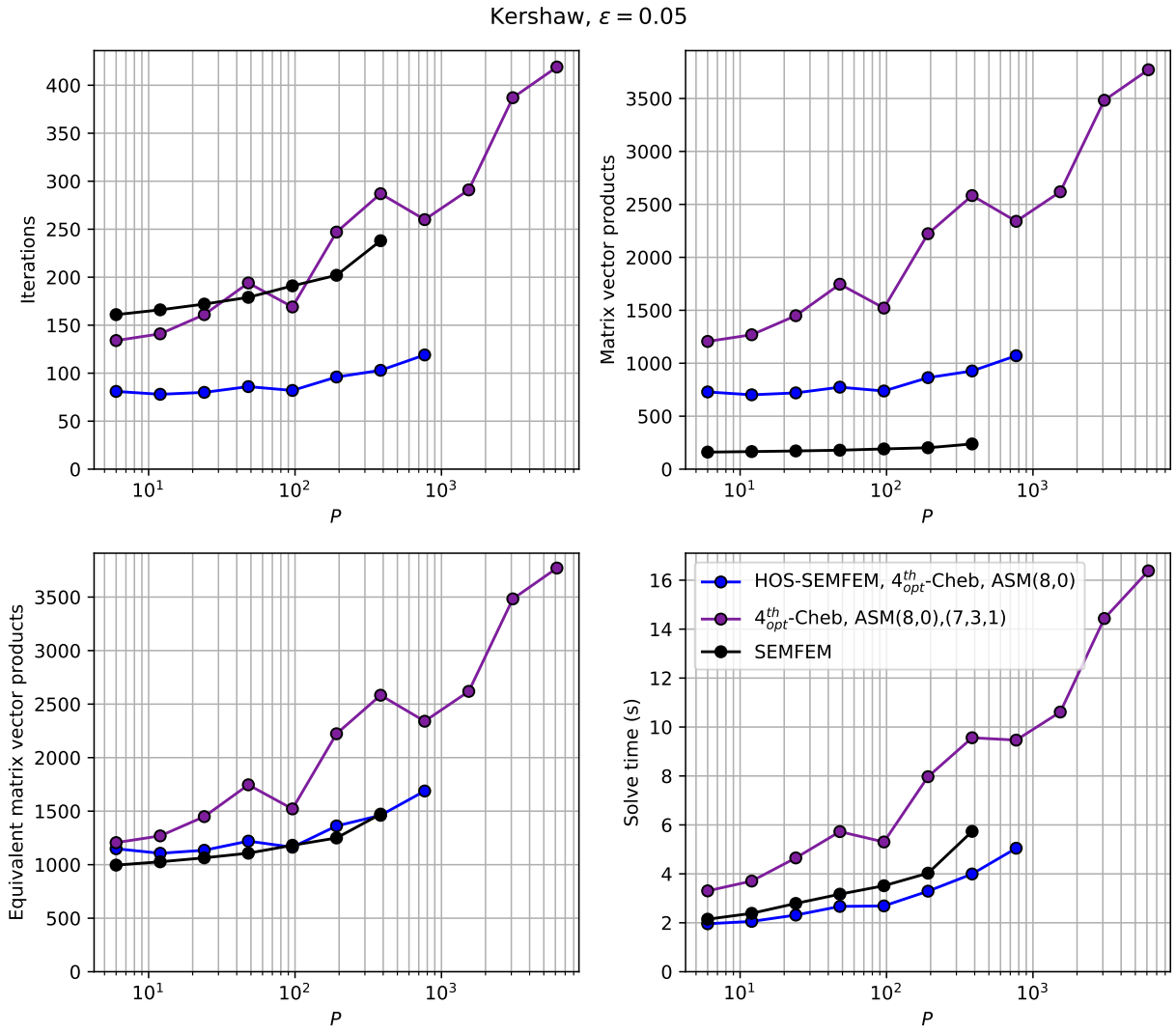



Figure 7.28: Weak scaling results for the Kershaw problem with $\varepsilon = 0.05$. The number of GPUs, $P$, is varied from $P = 6$ to $P = 6144$. The number of gridpoints per GPU is fixed to $n/P = 2.67M$. For a fair comparison, the best solver configuration for each case *at scale* is used. For simplicity, we consider $\omega_c = 1$ for the HOS-SEMFEM scheme.

iteration count increase for the larger problem sizes limits the weak scalability of the method. The HOS-SEMFEM and SEMFEM preconditioning approaches, on the other hand, exhibit a higher dependency on the number of GPUs. The reasons for this are two-fold: first, like the $p$-multigrid methods, the iteration count grows according problem size; second, the AMG V-cycle becomes more expensive to apply *per cycle* as the number of GPUs increases. This latter point is anticipated by the operator cost study from section 3.3.

# Chapter 8: Conclusions

## 8.1 SUMMARY AND IMPACT

In this thesis, we develop a suite of solver techniques for the solution of the SEM-discretized Poisson equation arising from the simulation of the incompressible Navier-Stokes. Through these solver techniques, we are able to greatly improve the time-to-solution of the incompressible Navier-Stokes equations in `nekRS`.

As part of this work, the Schwarz smoothers described in section 2.6 have been efficiently implemented on the GPU for `nekRS`. As described in section 3.2.2, the kernels associated with applying the Schwarz smoother are well tuned for both NVIDIA and AMD GPUs, thanks in large part to Tim Warburton's help on the `nekRS` project. These smoothers are then combined with the existing literature on Chebyshev polynomial smoothing, described in section 2.7. This itself is a contribution, as previous works typically only consider the use of point-wise Jacobi smoothing in the polynomial smoothers. These approaches are then combined with the geometric $p$-multigrid preconditioner to develop solvers that are able to achieve high parallel scalability on the GPU in section 2.9.

The Chebyshev polynomial smoothers are then further improved through the use of the fourth-kind and optimized fourth-kind Chebyshev polynomials, developed by James Lottes in [23], as described in chapter 4. These polynomials achieve provably optimal convergence rates, especially with high polynomial smoother orders. In particular, the multigrid error contraction factor using the standard Chebyshev polynomial smoothers only scale as $O(k^{-1})$. However, through the fourth-kind Chebyshev polynomials, $O(k^{-2})$ convergence rates are achieved. This is a significant improvement that can enable the use of higher-order smoother polynomials to more effectively control the coarse-grid solve cost associated with the multigrid V-cycle preconditioners.

Inspired by the theoretical results from [23], we develop a rigorous approach to selecting *how* to distribute smoothing passes across the pre- and post-smoothing phases of a multigrid V-cycle. This approach is guided by multi-level V-cycle error analysis provided through [23]. Putting all $2k$ smoothing passes in the pre-smoothing phase and omitting the post-smoothing phase is found to be optimal, provided that the multigrid approximation property constant is sufficiently large. Asymptotically, this approach *improves* the multigrid error contraction factor by 15% compared to the standard $k$ smoothing passes in the pre- and post-smoothing phases. The authors further demonstrate, through use of a computer algebra system in `sympy` [82], that either the symmetric $k$ pre- and post-smoothing passes or the $2k$

pre-smoothing passes with no post-smoothing is optimal in most cases for the Chebyshev polynomial smoothers.

A multigrid approach wherein the coarse-grid solve is treated in an additive fashion is proposed in chapter 5. While this approach is not found to be competitive with the standard multiplicative multigrid approach, techniques for improving the performance of the additive multigrid approach are discussed and left as future work in section 8.2.

Finally, we have developed a new auxiliary space method utilizing the low-order operator as a coarse-space. This method is described in chapter 6. Through this method, we are able to achieve a significant reduction in the number of iterations required to reach convergence at the expense of a few $p$-multigrid smoothing operations.

The improvements described in this thesis are not only relevant to `nekRS` and high-order spectral element methods, but also to the broader solver community as well. As part of this work, the fourth-kind and optimized fourth-kind Chebyshev smoothing polynomials discussed in chapter 4 have been implemented in `Trilinos/MueLu` [19], a multigrid preconditioner library for the Trilinos project [107]; `petsc`, the Portable, Extensible Toolkit for Scientific Computation [18]; `LFAToolkit.jl` [20], a library for the analysis of multigrid methods via local Fourier analysis; and soon `hypre` [17] in `https://github.com/hypre-space/hypre/pull/856`, a library for the solution of large, sparse linear systems arising from the discretization of PDEs.

## 8.2 FUTURE WORK

Through the course of this work, we have identified several promising avenues for future research. In this section, we describe these avenues for future work.

In section 3.3, we show that the coarse-grid solve component of the geometric $p$-multigrid method can hinder the parallel scalability of the solver. This coarse-grid solve, however, is approximated using algebraic multigrid techniques. In this thesis, not much attention is afforded to the algebraic multigrid solver settings, except that the settings described in section 2.4 are reasonably good. However, others have contributed much work on optimizing the performance of the algebraic multigrid solvers. For example, Zaman and coworkers [108] seek to optimize the NP-complete coarse-grid selection problem through simulated annealing. In [109], Katrutsa and coworkers consider the black-box learning of multigrid parameters. Huang and coworkers discuss learning optimal multigrid smoothers via neural networks in [110]. By optimizing the underlying algebraic multigrid solver through these techniques, the cost of the coarse-grid solve component of the geometric $p$-multigrid may be further reduced, improving the parallel scalability of the solver.

Another approach for the coarse-grid solve component is significantly simpler. In [111], Jansson and coworkers use a similar geometric $p$-multigrid preconditioner as described in section 2.6. However, instead of using an algebraic multigrid solver, they solve the coarse-grid problem using an inner preconditioned conjugate gradient solver. The conjugate gradient solver, however, requires many dot products if the iteration count is high. A potential way to mitigate this is to take fewer iterations with a high-quality Chebyshev polynomial preconditioner. In this approach, the same algorithm from alg. 2.12 can be used to accelerate Jacobi *as a preconditioner*. One notable difference, however, is that $\lambda_{min}$ in this case should be chosen as the *actual* smallest eigenvalue of the $D^{-1}A$ [31]. This allows for Chebyshev iterations with similar convergence to PCG to be utilized without requiring dot-products. Other polynomial preconditioners, such as the polynomial preconditioned GMRES algorithm discussed in [112], can be used to achieve similar results. One potential way to *increase* the arithmetic intensity of this operator is to also consider evaluating the Chebyshev polynomial in *parallel* (e.g., [113, 114]). This may be especially useful if several matrix-vector products can be "blocked" into a single GPU kernel launch, allowing the operator to be evaluated across many vectors at once.

In chapter 4, we consider the application of the symmetric pre- and post-smoother pass distributions, $(k, k)$, versus the asymmetric $(2k, 0)$ approach where the post-smoother is omitted. The *operation count* in terms of matrix-vector products and smoother applications are equivalent between the two approaches, making it possible to compare only the convergence rate when choosing between the two approaches. However, the repeated operations required to form the $2k$-th order polynomial in the $(2k, 0)$ approach may improve the execution time of the smoother due to cache reuse. On GPU-architectures, the scaling limit in terms of points per processor, $n/P$, is large at $n/P \sim 2M$. This unfortunately prevents the cache reuse effect from being present, as too much operator data is streamed in memory for this potential beneficial cache effect to be present. However, on CPU architectures, the cache reuse may allow the smoother to execute faster, making the cost of applying the $(2k, 0)$ approach less than the $(k, k)$ approach *per iteration*. When further combined with the potential convergence improvement, this makes the $(2k, 0)$ approach more attractive. Fully exploiting this potential benefit of the $(2k, 0)$ approach, specifically on CPU architectures, is left as future work.

In section 4.3, we extend Thompson's `LFAToolkit.jl` [20] to support the analysis of smoother polynomials built on the fourth-kind and optimized fourth-kind Chebyshev polynomials from chapter 4. These two polynomial smoothers are themselves optimal with respect to different error bounds. The fourth-kind Chebyshev polynomials are the solution to a *weighted* mini-max problem posed by the two-level error bound established by Hackbusch

[79] in eq. (4.1), while the optimized fourth-kind Chebyshev polynomials are the solution to the multilevel error bound due to Lottes [23] in lemma 4.1. As we observe that the local Fourier analysis (LFA) provides sharper estimates for multigrid error contraction rates, constructing an optimal polynomial smoother with respect to the LFA error bound may improve the performance of the smoother. While the polynomials constructed in section 4.5 are optimal with respect to the solver convergence rate, assuming the Nelder-Mead optimizer is able to find the global minimum, the LFA error bound may provide smoother polynomials that are effective without requiring the setup costs incurred in section 4.5.

Section 5.2 describes the details for a multigrid scheme in which *only* the coarse-grid solve is treated in an additive fashion. While this method proves ineffective in section 7.1, possible improvements may be made by ensuring that each component of the additive multigrid scheme occupies the same amount of time. For example, if the coarse-grid solve is cheaper than the remainder of the V-cycle, then the quality of the coarse-grid can be improved through increasing the AMG smoother order, for example. This can be done at *no additional cost per iteration*, and may help the convergence rate. Further, the problem scales considered in section 7.1 are relatively small. Larger scale problems wherein the coarse-grid solve cost is more significant may benefit from the additive coarse-grid solve approach.

## References


[1] "Top500: June," https://top500.org/lists/top500/2023/06/, accessed: 2023-06-03.

[2] M. Deville, P. Fischer, and E. Mund, *High-order methods for incompressible fluid flow*. Cambridge: Cambridge University Press, 2002.

[3] P. Fischer, J. Lottes, and H. Tufo, "Nek5000," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2007.

[4] P. Fischer, S. Kerkemeier, M. Min, Y.-H. Lan, M. Phillips, T. Rathnayake, E. Merzari, A. Tomboulides, A. Karakus, N. Chalmers et al., "Nekrs, a gpu-accelerated spectral element navier–stokes solver," *Parallel Computing*, vol. 114, p. 102982, 2022.

[5] C. Canuto, P. Gervasio, and A. Quarteroni, "Finite-element preconditioning of g-ni spectral methods," *SIAM Journal on Scientific Computing*, vol. 31, no. 6, pp. 4422–4451, 2010.

[6] S. A. Orszag, "Spectral methods for problems in complex geometrics," in *Numerical methods for partial differential equations*. Elsevier, 1979, pp. 273–305.

[7] H. Sundar, G. Stadler, and G. Biros, "Comparison of multigrid algorithms for high-order continuous finite element discretizations," *Numerical Linear Algebra with Applications*, vol. 22, no. 4, pp. 664–680, 2015.

[8] M. Adams, M. Brezina, J. Hu, and R. Tuminaro, "Parallel multigrid smoothing: polynomial versus gauss–seidel," *Journal of Computational Physics*, vol. 188, no. 2, pp. 593–610, 2003.

[9] M. Kronbichler and K. Ljungkvist, "Multigrid for matrix-free high-order finite element computations on graphics processors," *ACM Transactions on Parallel Computing*, vol. 6, no. 1, pp. 1–32, 2019.

[10] J. W. Lottes and P. F. Fischer, "Hybrid multigrid/schwarz algorithms for the spectral element method," *Journal of Scientific Computing*, vol. 24, no. 1, pp. 45–78, 2005.

[11] J. Stiller, "Nonuniformly weighted schwarz smoothers for spectral element multigrid," *Journal of Scientific Computing*, vol. 72, no. 1, pp. 81–96, 2017.

[12] S. Loisel, R. Nabben, and D. B. Szyld, "On hybrid multigrid-schwarz algorithms," *Journal of Scientific Computing*, vol. 36, no. 2, pp. 165–175, 2008.

[13] L. Olson, "Algebraic multigrid preconditioning of high-order spectral elements for elliptic problems on a simplicial mesh," *SIAM Journal on Scientific Computing*, vol. 29, no. 5, pp. 2189–2209, 2007.

[14] P. D. Bello-Maldonado and P. F. Fischer, "Scalable low-order finite element preconditioners for high-order spectral element poisson solvers," *SIAM Journal on Scientific Computing*, vol. 41, no. 5, pp. S2–S18, 2019.

[15] W. Pazner, T. Kolev, and J.-S. Camier, "End-to-end gpu acceleration of low-order-refined preconditioning for high-order finite element discretizations," *arXiv preprint arXiv:2210.12253*, 2022.

[16] M. Min, Y.-H. Lan, P. Fischer, E. Merzari, S. Kerkemeier, M. Phillips, T. Rathnayake, A. Novak, D. Gaston, N. Chalmers et al., "Optimization of full-core reactor simulations on summit," in *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2022, pp. 1067–1077.

[17] "*hypre*: High performance preconditioners," https://llnl.gov/casc/hypre, https://github.com/hypre-space/hypre.

[18] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp et al., "Petsc users manual," 2019.

[19] A. Prokopenko, J. Hu, T. Wiesner, C. Siefert, and R. Tuminaro, "Muelu user's guide 1.0 (trilinos version 11.12)," *SAND2014-18874. Oct*, 2014.

[20] J. L. Thompson, "LFAToolkit," https://github.com/jeremylt/LFAToolkit.jl, 2021.

[21] J. H. Bramble, J. E. Pasciak, and J. Xu, "Parallel multilevel preconditioners," *Mathematics of computation*, vol. 55, no. 191, pp. 1–22, 1990.

[22] P. S. Vassilevski and U. M. Yang, "Reducing communication in algebraic multigrid using additive variants," *Numerical Linear Algebra with Applications*, vol. 21, no. 2, pp. 275–296, 2014.

[23] J. Lottes, "Optimal polynomial smoothers for multigrid v-cycles," *preprint arXiv:2202.08830*, 2022.

[24] J. Xu, "The auxiliary space method and optimal multigrid preconditioning techniques for unstructured grids," *Computing*, vol. 56, no. 3, pp. 215–235, 1996.

[25] M. Phillips, S. Kerkemeier, and P. Fischer, "Tuning spectral element preconditioners for parallel scalability on gpus," in *Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 2022, pp. 37–48.

[26] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev et al., "Mfem: A modular finite element methods library," *Computers & Mathematics with Applications*, vol. 81, pp. 42–74, 2021.

[27] J. Brown, A. Abdelfattah, V. Barra, N. Beams, J.-S. Camier, V. Dobrev, Y. Dudouit, L. Ghaffari, T. Kolev, D. Medina et al., "libceed: Fast algebra for high-order element-based discretizations," *Journal of Open Source Software*, vol. 6, no. 63, p. 2945, 2021.

[28] Y. Notay, "Flexible conjugate gradients," *SIAM Journal on Scientific Computing*, vol. 22, no. 4, pp. 1444–1460, 2000.

[29] G. H. Golub and Q. Ye, "Inexact preconditioned conjugate gradient method with inner-outer iteration," *SIAM Journal on Scientific Computing*, vol. 21, no. 4, pp. 1305–1320, 1999.

[30] H. Bouwmeester, A. Dougherty, and A. V. Knyazev, "Nonsymmetric preconditioning for conjugate gradient and steepest descent methods," *Procedia Computer Science*, vol. 51, pp. 276–285, 2015.

[31] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[32] Y. Saad, "A flexible inner-outer preconditioned gmres algorithm," *SIAM Journal on Scientific Computing*, vol. 14, no. 2, pp. 461–469, 1993.

[33] M. Embree, "How descriptive are gmres convergence bounds?" *arXiv preprint arXiv:2209.01231*, 2022.

[34] N. N. Abdelmalek, "Round off error analysis for gram-schmidt method and solution of linear least squares problems," *BIT Numerical Mathematics*, vol. 11, no. 4, pp. 345–367, 1971.

[35] L. Giraud, J. Langou, and M. Rozloznik, "The loss of orthogonality in the gram-schmidt orthogonalization process," *Computers & Mathematics with Applications*, vol. 50, no. 7, pp. 1069–1075, 2005.

[36] S. Thomas, E. Carson, M. Rozložník, A. Carr, and K. Świrydowicz, "Iterated-gauss-seidel gmres," *arXiv preprint arXiv:2205.07805*, 2022.

[37] A. Greenbaum, M. Rozložnik, and Z. Strakoš, "Numerical behaviour of the modified gram-schmidt gmres implementation," *BIT Numerical Mathematics*, vol. 37, no. 3, pp. 706–719, 1997.

[38] P. F. Fischer, "Projection techniques for iterative solution of ax= b with successive right-hand sides," *Computer methods in applied mechanics and engineering*, vol. 163, no. 1-4, pp. 193–204, 1998.

[39] M. L. Parks, E. De Sturler, G. Mackey, D. D. Johnson, and S. Maiti, "Recycling krylov subspaces for sequences of linear systems," *SIAM Journal on Scientific Computing*, vol. 28, no. 5, pp. 1651–1674, 2006.

[40] N. Christensen, "Efficient projection space updates for the approximation of iterative solutions to linear systems with successive right hand sides," M.S. thesis, University of Illinois Urbana Champaign, 2017.

[41] A. P. Austin, N. Chalmers, and T. Warburton, "Initial guesses for sequences of linear systems in a gpu-accelerated incompressible flow solver," *SIAM Journal on Scientific Computing*, vol. 43, no. 4, pp. C259–C289, 2021.

[42] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh et al., "Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods," *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. S602–S626, 2015.

[43] U. M. Yang et al., "Boomeramg: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002.

[44] P. F. Fischer, "An overlapping schwarz method for spectral element solution of the incompressible navier–stokes equations," *Journal of Computational Physics*, vol. 133, no. 1, pp. 84–101, 1997.

[45] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou, "Accelerate gpu concurrent kernel execution by mitigating memory pipeline stalls," in *2018 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2018, pp. 208–220.

[46] S. McCormick, "Multigrid methods for variational problems: general theory for the v-cycle," *SIAM Journal on Numerical Analysis*, vol. 22, no. 4, pp. 634–643, 1985.

[47] J. L. Thompson, J. Brown, and Y. He, "Local fourier analysis of p-multigrid for high-order finite element operators," *arXiv preprint arXiv:2108.01751*, 2021.

[48] X.-C. Cai and M. Sarkis, "A restricted additive schwarz preconditioner for general sparse linear systems," *Siam journal on scientific computing*, vol. 21, no. 2, pp. 792–797, 1999.

[49] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Multigrid smoothers for ultraparallel computing," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2864–2887, 2011.

[50] V. T. Zhukov, N. D. Novikova, and O. B. Feodoritova, "Multigrid method for elliptic equations with anisotropic discontinuous coefficients," *Computational Mathematics and Mathematical Physics*, vol. 55, no. 7, pp. 1150–1163, 2015.

[51] T. Kolev, P. Fischer, A. P. Austin, A. T. Barker, N. Beams, J. Brown, J.-S. Camier, N. Chalmers, V. Dobrev, Y. Dudouit, L. Ghaffari, S. Kerkemeier, Y.-H. Lan, E. Merzari, M. Min, W. Pazner, T. Ratnayaka, M. S. Shephard, M. H. Siboni, C. W. Smith, J. L. Thompson, S. Tomov, and T. Warburton, "CEED ECP Milestone Report: High-order algorithmic developments and optimizations for large-scale GPU-accelerated simulations," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4672664

[52] D. S. Kershaw, "Differencing of the diffusion equation in lagrangian hydrodynamic codes," *Journal of Computational Physics*, vol. 39, no. 2, pp. 375–395, 1981.

[53] Y.-H. Lan, P. Fischer, E. Merzari, and M. Min, "All-hex meshing strategies for densely packed spheres," *preprint arXiv:2106.00196*, 2021.

[54] D. Reger, E. Merzari, H. Yuan, S. King, Y. Hassan, K. Ngo, P. Balestra, and S. Schunert, "Large eddy simulation of a 67-pebble bed experiment," in *Advances in Thermal-Hydraulics*, June 2022.

[55] D. Reger, E. Merzari, P. Balestra, S. Schunert, Y. Hassan, and H. Yuan, "Toward development of an improved friction correlation for the near-wall region of pebble bed systems," *preprint arXiv:2203.05041*, 2022.

[56] M. L. Shur, P. R. Spalart, M. K. Strelets, and A. K. Travin, "Direct numerical simulation of the two-dimensional speed bump flow at increasing reynolds numbers," *International Journal of Heat and Fluid Flow*, vol. 90, p. 108840, 2021.

[57] Y. Notay, "Convergence analysis of perturbed two-grid and multigrid methods," *SIAM journal on numerical analysis*, vol. 45, no. 3, pp. 1035–1044, 2007.

[58] K. Mittal and P. Fischer, "Mesh smoothing for the spectral element method," *Journal of Scientific Computing*, vol. 78, pp. 1152–1173, 2019.

[59] A. Imakura, T. Sakurai, K. Sumiyoshi, and H. Matsufuru, "An auto-tuning technique of the weighted jacobi-type iteration used for preconditioners of krylov subspace methods," in *2012 IEEE 6th International Symposium on Embedded Multicore SoCs*. IEEE, 2012, pp. 183–190.

[60] K. Yamada, T. Katagiri, H. Takizawa, K. Minami, M. Yokokawa, T. Nagai, and M. Ogino, "Preconditioner auto-tuning using deep learning for sparse iterative algorithms," in *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE, 2018, pp. 257–262.

[61] J. Brown, Y. He, S. MacLachlan, M. Menickelly, and S. M. Wild, "Tuning multigrid methods with robust optimization and local fourier analysis," *SIAM Journal on Scientific Computing*, vol. 43, no. 1, pp. A109–A138, 2021.

[62] D. S. Medina, A. St-Cyr, and T. Warburton, "Occa: A unified approach to multithreading languages," *preprint arXiv:1403.0968*, 2014.

[63] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: accelerating small matrix multiplications by runtime code generation," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 981–991.

[64] B. Hess, J. Gong, S. Páll, P. Schlatter, and A. Peplinski, "Highly tuned small matrix multiplications applied to spectral element code nek5000," 2016.

[65] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies," *IBM Reserach Report, RC24704 (W0812-047)*, 2008.

[66] M. Ashoury, M. Loni, F. Khunjush, and M. Daneshtalab, "Auto-spmv: Automated optimizing spmv kernels on gpu," *arXiv preprint arXiv:2302.05662*, 2023.

[67] A. Klöckner, "Loo. py: transformation-based code generation for gpus and cpus," in *Proceedings of ACM SIGPLAN international workshop on libraries, languages, and compilers for array programming*, 2014, pp. 82–87.

[68] A. Klöckner, L. C. Wilcox, and T. Warburton, "Array program transformation with loo. py by example: high-order finite elements," in *Proceedings of the 3rd ACM SIGPLAN international workshop on libraries, languages, and compilers for array programming*, 2016, pp. 9–16.

[69] N. Chalmers, A. Karakus, A. P. Austin, K. Swirydowicz, and T. Warburton, "libParanumal: a performance portable high-order finite element library."

[70] K. Świrydowicz, N. Chalmers, A. Karakus, and T. Warburton, "Acceleration of tensor-product operations for high-order finite element methods," *The International Journal of High Performance Computing Applications*, vol. 33, no. 4, pp. 735–757, 2019.

[71] J. Brown, V. Barra, N. Beams, L. Ghaffari, M. Knepley, W. Moses, R. Shakeri, K. Stengel, J. L. Thompson, and J. Zhang, "Performance portable solid mechanics via matrix-free $p$-multigrid," *arXiv preprint arXiv:2204.01722*, 2022.

[72] N. Chalmers, A. Mishra, D. McDougall, and T. Warburton, "Hipbone: A performance-portable graphics processing unit-accelerated c++ version of the nekbone benchmark," *The International Journal of High Performance Computing Applications*, p. 10943420231178552, 2023.

[73] N. Chalmers and T. Warburton, "Portable high-order finite element kernels i: Streaming operations," *arXiv preprint arXiv:2009.10917*, 2020.

[74] P. F. Fischer, "Scaling limits for pde-based simulation," in *22nd AIAA Computational Fluid Dynamics Conference*, 2015, p. 3049.

[75] P. D. Bello-Maldonado, "Polynomial reduction with full domain decomposition preconditioner for spectral element poisson solvers," Ph.D. dissertation, University of Illinois Urbana Champaign, 2022.

[76] A. Bienz, L. N. Olson, W. D. Gropp, and S. Lockhart, "Modeling data movement performance on heterogeneous architectures," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7.

[77] P. Fischer, M. Min, T. Rathnayake, S. Dutta, T. Kolev, V. Dobrev, J.-S. Camier, M. Kronbichler, T. Warburton, K. Świrydowicz et al., "Scalability of high-performance pde solvers," *The International Journal of High Performance Computing Applications*, vol. 34, no. 5, pp. 562–586, 2020.

[78] M. Phillips and P. Fischer, "Optimal chebyshev smoothers and one-sided v-cycles," *arXiv preprint arXiv:2210.03179*, 2022.

[79] W. Hackbusch, "Multi-grid convergence theory," in *Multigrid methods*. Springer, 1982, pp. 177–219.

[80] J. C. Mason, "Chebyshev polynomials of the second, third and fourth kinds in approximation, indefinite integration, and integral transforms," *Journal of Computational and Applied Mathematics*, vol. 49, no. 1-3, pp. 169–178, 1993.

[81] R. D. Falgout, R. Li, B. Sjögreen, L. Wang, and U. M. Yang, "Porting hypre to heterogeneous computer architectures: Strategies and experiences," *Parallel Computing*, vol. 108, p. 102840, 2021.

[82] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh et al., "Sympy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, p. e103, 2017.

[83] K. Stuben, "Algebraic multigrid (amg): an introduction with applications," *GMD report*, 1999.

[84] R. D. Falgout and J. B. Schroder, "Non-galerkin coarse grids for algebraic multigrid," *SIAM Journal on Scientific Computing*, vol. 36, no. 3, pp. C309–C334, 2014.

[85] A. Bienz, W. D. Gropp, and L. N. Olson, "Reducing communication in algebraic multigrid with multi-step node aware communication," *The International Journal of High Performance Computing Applications*, vol. 34, no. 5, pp. 547–561, 2020.

[86] A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Mathematics of Computation*, vol. 31, no. 138, pp. 333–390, 1977.

[87] R. Wienands and W. Joppich, *Practical Fourier analysis for multigrid methods*. CRC press, 2004.

[88] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence properties of the nelder–mead simplex method in low dimensions," *SIAM Journal on optimization*, vol. 9, no. 1, pp. 112–147, 1998.

[89] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[90] R. O'Neill, "Algorithm as 47: Function minimization using a simplex procedure," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 20, no. 3, pp. 338–345, 1971.

[91] J. Burkhardt, "Asa047: Nelder-mead minimization algorithm," *C++ library*, 2008.

[92] A. Greenbaum, "A multigrid method for multiprocessors," *Applied mathematics and computation*, vol. 19, no. 1-4, pp. 75–88, 1986.

[93] J. Wolfson-Pou and E. Chow, "Asynchronous multigrid methods," in *2019 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2019, pp. 101–110.

[94] Y. Notay and A. Napov, "Further comparison of additive and multiplicative coarse grid correction," *Applied Numerical Mathematics*, vol. 65, pp. 53–62, 2013.

[95] OpenMP Architecture Review Board, "OpenMP application program interface version 5.2," Nov. 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[96] W. Gropp and R. Thakur, "Issues in developing a thread-safe mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 13th European PVM/MPI User's Group Meeting Bonn, Germany, September 17-20, 2006 Proceedings 13*. Springer, 2006, pp. 12–21.

[97] T. Patinyasakdikul, D. Eberius, G. Bosilca, and N. Hjelm, "Give mpi threading a fair chance: A study of multithreaded mpi designs," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–11.

[98] R. Zambre and A. Chandramowlishwaran, "Lessons learned on mpi+ threads communication," *arXiv preprint arXiv:2206.14285*, 2022.

[99] S. Nepomnyaschikh, *Decomposition and fictitious domains methods for elliptic boundary value problems*. New York University. Courant Institute of Mathematical Sciences. Computer . . . , 1991.

[100] R. Hiptmair and J. Xu, "Nodal auxiliary space preconditioning in h (curl) and h (div) spaces," *SIAM Journal on Numerical Analysis*, vol. 45, no. 6, pp. 2483–2509, 2007.

[101] T. V. Kolev, J. E. Pasciak, and P. S. Vassilevski, "H (curl) auxiliary mesh preconditioning," *Numerical Linear Algebra with Applications*, vol. 15, no. 5, pp. 455–471, 2008.

[102] T. V. Kolev and P. S. Vassilevski, "Parallel auxiliary space amg for h (curl) problems," *Journal of Computational Mathematics*, pp. 604–623, 2009.

[103] T. V. Kolev and P. S. Vassilevski, "Parallel auxiliary space amg solver for h(div) problems," *SIAM Journal on Scientific Computing*, vol. 34, no. 6, pp. A3079–A3098, 2012.

[104] A. T. Barker and T. Kolev, "Matrix-free preconditioning for high-order h (curl) discretizations," *Numerical Linear Algebra with Applications*, vol. 28, no. 2, p. e2348, 2021.

[105] J. Xu and L. Zikatanov, "Algebraic multigrid methods," *Acta Numerica*, vol. 26, pp. 591–721, 2017.

[106] A. Bienz, R. D. Falgout, W. Gropp, L. N. Olson, and J. B. Schroder, "Reducing parallel communication in algebraic multigrid through sparsification," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S332–S357, 2016.

[107] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps et al., "An overview of the trilinos project," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 397–423, 2005.

[108] T. U. Zaman, S. P. MacLachlan, L. N. Olson, and M. West, "Coarse-grid selection using simulated annealing," *Journal of Computational and Applied Mathematics*, vol. 431, p. 115263, 2023.

[109] A. Katrutsa, T. Daulbaev, and I. Oseledets, "Black-box learning of multigrid parameters," *Journal of Computational and Applied Mathematics*, vol. 368, p. 112524, 2020.

[110] R. Huang, R. Li, and Y. Xi, "Learning optimal multigrid smoothers via neural networks," *SIAM Journal on Scientific Computing*, no. 0, pp. S199–S225, 2022.

[111] N. Jansson, M. Karp, A. Podobas, S. Markidis, and P. Schlatter, "Neko: A modern, portable, and scalable framework for high-fidelity computational fluid dynamics," *arXiv preprint arXiv:2107.01243*, 2021.

[112] J. A. Loe, H. K. Thornquist, and E. G. Boman, "Polynomial preconditioned gmres in trilinos: Practical considerations for high-performance computing," in *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 2020, pp. 35–45.

[113] R. Barrio and J. Sabadell, "A parallel algorithm to evaluate chebyshev series on a message passing environment," *SIAM Journal on Scientific Computing*, vol. 20, no. 3, pp. 964–969, 1998.

[114] A. M. Atallah and A. B. Younes, "Parallel evaluation of chebyshev approximations: Applications in astrodynamics," *The Journal of the Astronautical Sciences*, vol. 69, no. 3, pp. 692–717, 2022.

## Appendix A: Optimized 4th-Kind Chebyshev Tabulated Coefficients

Table A.1: Tabulated values of $\beta$ for the optimized 4th-kind Chebyshev smoother.

| $k$ | $\beta_i^{(k)}$ |
|---|---|
| 1 | 1.12500000000000 |
| 2 | 1.02387287570313 |
| | 1.26408905371085 |
| 3 | 1.00842544782028 |
| | 1.08867839208730 |
| | 1.33753125909618 |
| 4 | 1.00391310427285 |
| | 1.04035811188593 |
| | 1.14863498546254 |
| | 1.38268869241000 |
| 5 | 1.00212930146164 |
| | 1.02173711549260 |
| | 1.07872433192603 |
| | 1.19810065292663 |
| | 1.41322542791682 |
| 6 | 1.00128517255940 |
| | 1.01304293035233 |
| | 1.04678215124113 |
| | 1.11616489419675 |
| | 1.23829020218444 |
| | 1.43524297106744 |
| 7 | 1.00083464397912 |
| | 1.00843949430122 |
| | 1.03008707768713 |
| | 1.07408384092003 |
| | 1.15036186707366 |
| | 1.27116474046139 |
| | 1.45186658649364 |
| 8 | 1.00057246631197 |
| | 1.00577427662415 |
| | 1.02050187922941 |
| | 1.05019803444565 |
| | 1.10115572984941 |
| | 1.18086042806856 |
| | 1.29838585382576 |
| | 1.46486073151099 |
| 9 | 1.00040960072832 |
| | 1.00412439506106 |
| | 1.01460212148266 |
| | 1.03561113626671 |
| | 1.07139972529194 |
| | 1.12688273710962 |
| | 1.20785219140729 |
| | 1.32121930716746 |
| | 1.47529642820699 |
| 10 | 1.00030312229652 |
| | 1.00304840660796 |
| | 1.01077022715387 |
| | 1.02619011597640 |
| | 1.05231724933755 |
| | 1.09255743207549 |
| | 1.15083376663972 |
| | 1.23172250870894 |
| | 1.34060802024460 |
| | 1.48386124407011 |
| 11 | 1.00023058595209 |
| | 1.00231675024028 |
| | 1.00817245396304 |
| | 1.01982986566342 |
| | 1.03950210235324 |
| | 1.06965042700541 |
| | 1.11305754295742 |
| | 1.17290876275564 |
| | 1.25288300576792 |
| | 1.35725579919519 |
| | 1.49101672564139 |
| 12 | 1.00017947200828 |
| | 1.00180189139619 |
| | 1.00634861907307 |
| | 1.01537864566306 |
| | 1.03056942830760 |
| | 1.05376019693943 |
| | 1.08699862592072 |
| | 1.13259183097913 |
| | 1.19316273358172 |
| | 1.27171293675110 |
| | 1.37169337969799 |
| | 1.49708418575562 |
| 13 | 1.00014241921559 |
| | 1.00142906932629 |
| | 1.00503028986298 |
| | 1.01216910518495 |
| | 1.02414874342792 |
| | 1.04238158880820 |
| | 1.06842008128700 |
| | 1.10399010936759 |
| | 1.15102748242645 |
| | 1.21171811910125 |
| | 1.28854264865128 |
| | 1.38432619380991 |
| | 1.50229418757368 |
| 14 | 1.00011490538261 |
| | 1.00115246376914 |
| | 1.00405357333264 |
| | 1.00979590573153 |
| | 1.01941300472994 |
| | 1.03401425035436 |
| | 1.05480599606629 |
| | 1.08311420301813 |
| | 1.12040891660892 |
| | 1.16833095655446 |
| | 1.22872122288238 |
| | 1.30365305707817 |
| | 1.39546814053678 |
| | 1.50681646209583 |
| 15 | 1.00009404750752 |
| | 1.00094291696343 |
| | 1.00331449056444 |
| | 1.00800294833816 |
| | 1.01584236259140 |
| | 1.02772083317705 |
| | 1.04459535422831 |
| | 1.06750761206125 |
| | 1.09760092545889 |
| | 1.13613855366157 |
| | 1.18452361426236 |
| | 1.24432087304475 |
| | 1.31728069083392 |
| | 1.40536543893560 |
| | 1.51077872501845 |
| 16 | 1.00007794828179 |
| | 1.00078126847253 |
| | 1.00274487974401 |
| | 1.00662291017015 |
| | 1.01309858836971 |
| | 1.02289448329337 |
| | 1.03678321409983 |
| | 1.05559875719896 |
| | 1.08024848405560 |
| | 1.11172607131497 |
| | 1.15112543431072 |
| | 1.19965584614973 |
| | 1.25865841744946 |
| | 1.32962412656664 |
| | 1.41421360695576 |
| | 1.51427891730346 |

# Appendix B: Additional Results

## B.1 OVERLAPPING COARSE-GRID SOLVES

Supplementary results for section 7.1 are presented in this appendix.

### B.1.1 Kershaw

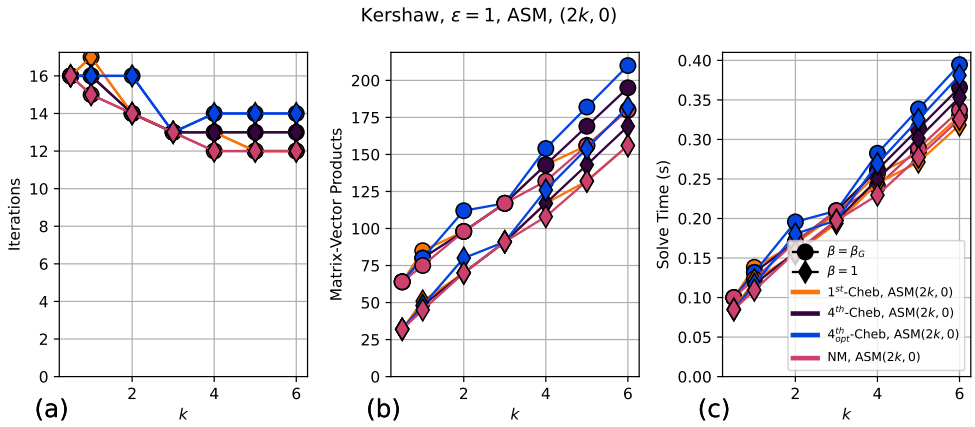



Figure B.1: Comparison of $\beta = 1$ and $\beta = \beta_G$. Kershaw, $\varepsilon = 1$. $E = 36^3, p = 7$. One Summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. Post smoothing is omitted, e.g. the $(2k, 0)$ approach. The smoother polynomial order, $k$, is varied.

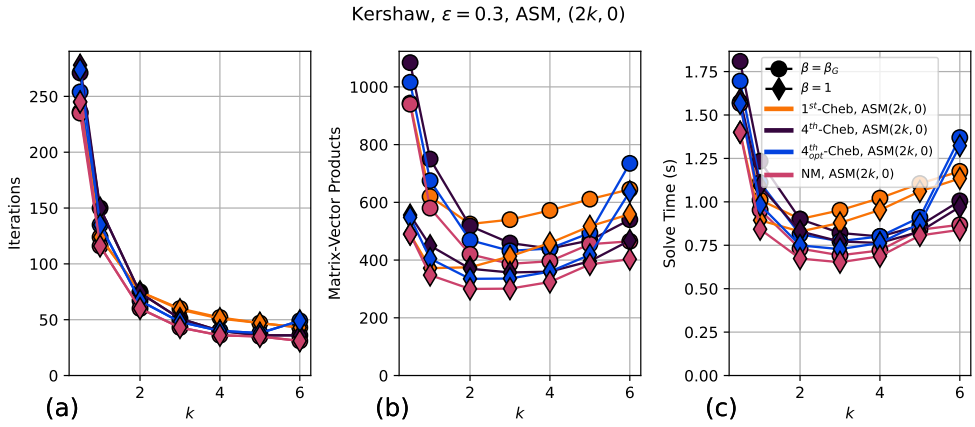



Figure B.2: Comparison of $\beta = 1$ and $\beta = \beta_G$. Kershaw, $\varepsilon = 0.3$. $E = 36^3, p = 7$. One Summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. Post smoothing is omitted, e.g. the $(2k, 0)$ approach. The smoother polynomial order, $k$, is varied.

A comparison of the $\beta = 1$ and $\beta = \beta_G$ coarse-grid correction factors are presented in figs. B.1 and B.2 for the $(2k, 0)$ V-cycle approach from chapter 4.

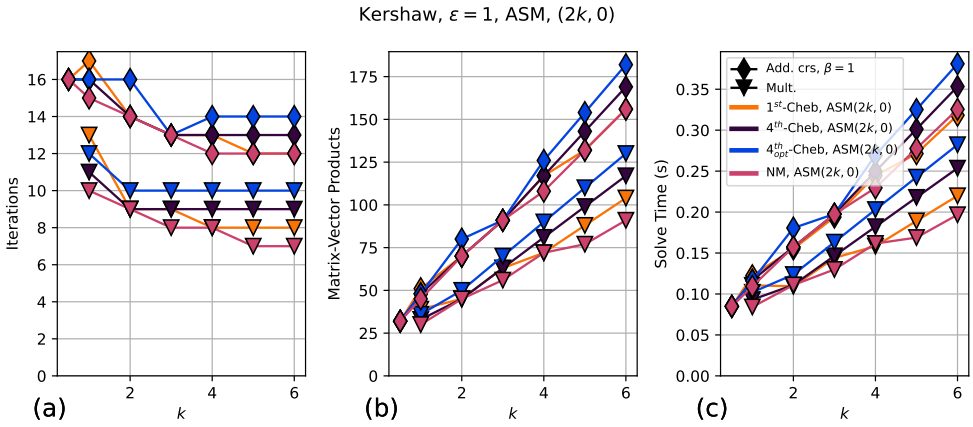



Figure B.3: Comparison of additive coarse-grid method with $\beta = 1$ the standard, multiplicative pMG method. Kershaw, $\varepsilon = 1$. $E = 36^3, p = 7$. One Summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. Post smoothing is omitted, e.g. the $(2k, 0)$ approach. The smoother polynomial order, $k$, is varied.

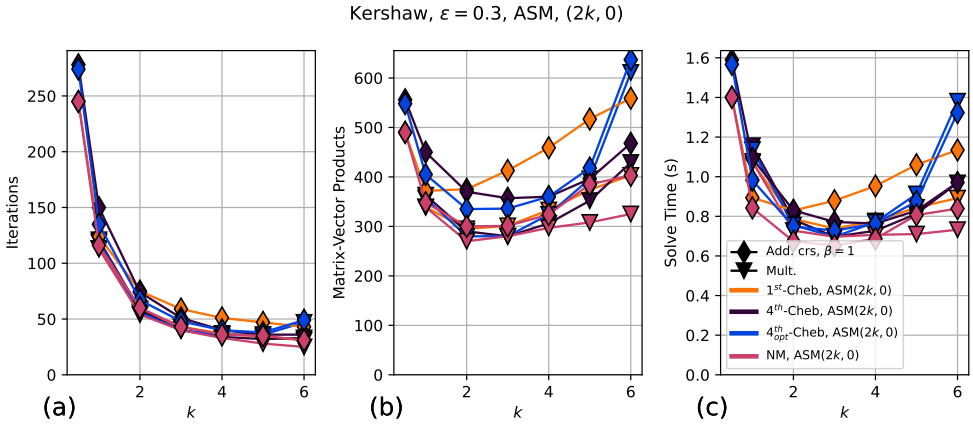



Figure B.4: Comparison of additive coarse-grid method with $\beta = 1$ the standard, multiplicative pMG method. Kershaw, $\varepsilon = 0.3$. $E = 36^3, p = 7$. One Summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. Post smoothing is omitted, e.g. the $(2k, 0)$ approach. The smoother polynomial order, $k$, is varied.

Figures B.3 and B.4 shows a performance comparison of the additive coarse-grid method with $\beta = 1$ to the standard, multiplicative pMG method for the $(2k, 0)$ V-cycle approach.

### B.1.2 Pebble Bed Cases

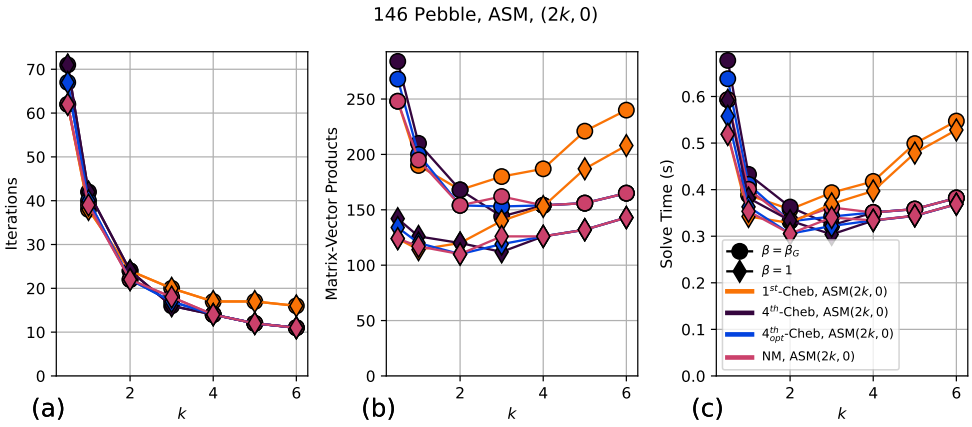



Figure B.5: 146 pebble case (fig. 2.11a). Comparison of $\beta = 1$ and $\beta = \beta_G$ for the additive coarse-grid method. A single summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 3.5M$. The polynomial smoother order, $k$, is varied. $(2k, 0)$ smoothing is used.

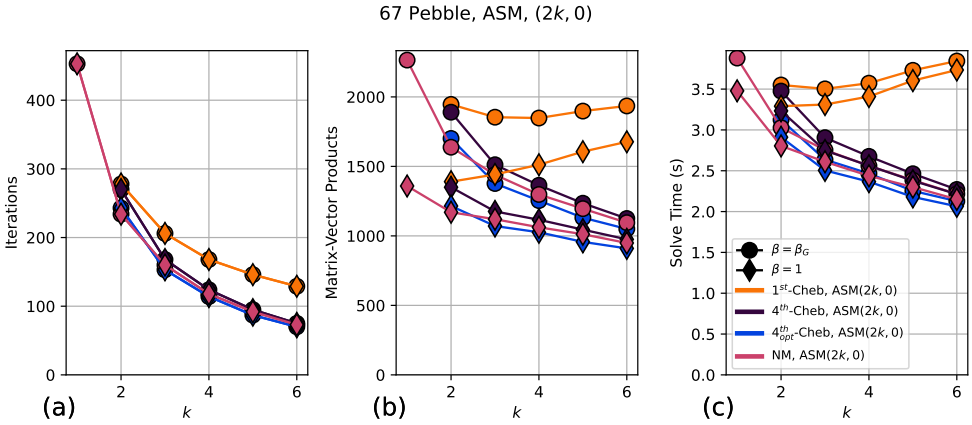



Figure B.6: 67 pebble case (fig. 2.11c). Comparison of $\beta = 1$ and $\beta = \beta_G$ for the additive coarse-grid method. Three summit nodes ($P = 18$ V100 GPUs) is used with $n/P \sim 2.33M$. The polynomial smoother order, $k$, is varied. $(2k, 0)$ smoothing is used.

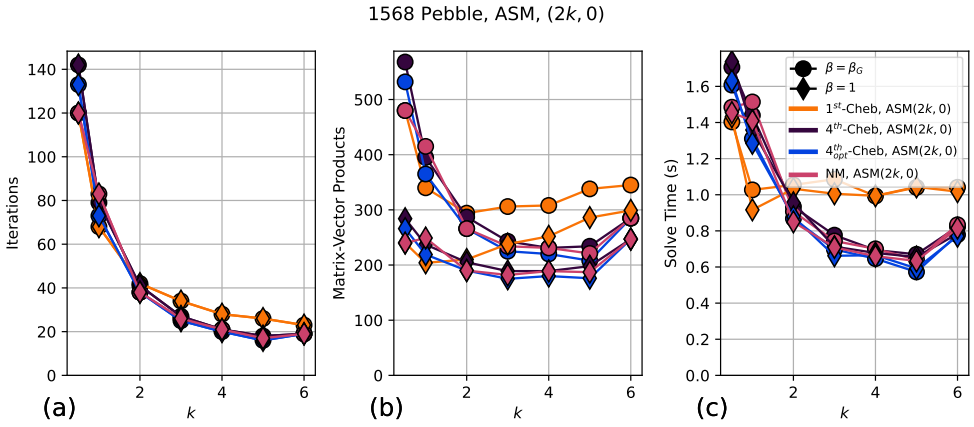



Figure B.7: 1568 pebble case (fig. 2.11b). Comparison of $\beta = 1$ and $\beta = \beta_G$ for the additive coarse-grid method. 12 Summit nodes ($P = 72$ V100 GPUs) is used with $n/P \sim 2.5M$. The polynomial smoother order, $k$, is varied. $(2k, 0)$ smoothing is used.

Figures B.5 to B.7 show solver performance comparisons of the $\beta = 1$ and $\beta = \beta_G$ coarse-grid correction factors for the additive V-cycle.

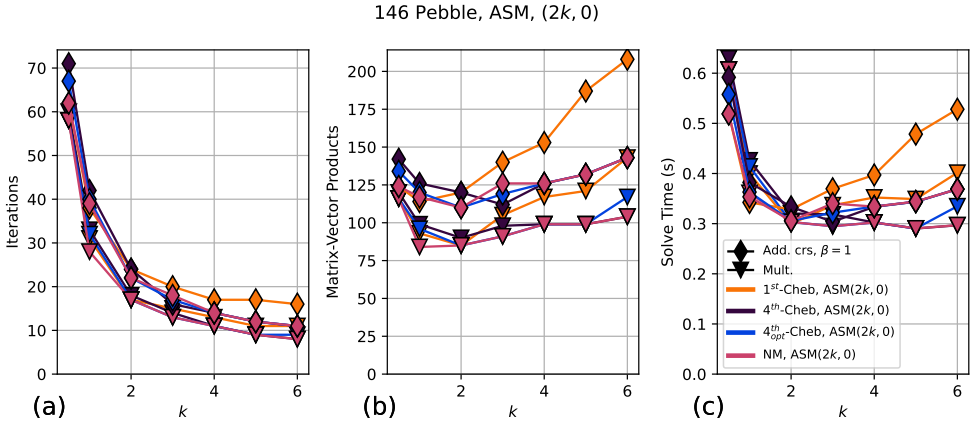



Figure B.8: 146 pebble case (fig. 2.11a). Comparison of additive coarse-grid solve with $\beta = 1$ and multiplicative V-cycle. A single summit node ($P = 6$ V100 GPUs) is used with $n/P \sim 3.5M$. The polynomial smoother order, $k$, is varied. $(2k, 0)$ smoothing is used.

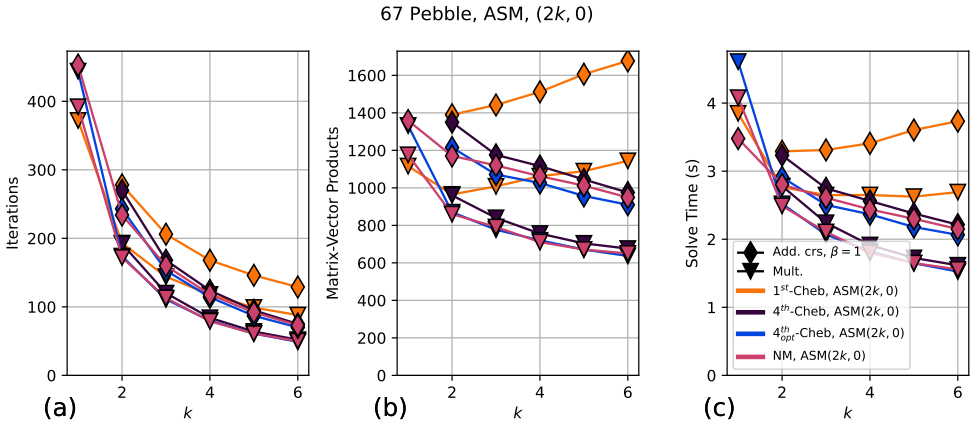



Figure B.9: 67 pebble case (fig. 2.11c). Comparison of additive coarse-grid solve with $\beta = 1$ and multiplicative V-cycle. Three summit nodes ($P = 18$ V100 GPUs) is used with $n/P \sim 2.33M$. The polynomial smoother order, $k$, is varied. $(2k, 0)$ smoothing is used.

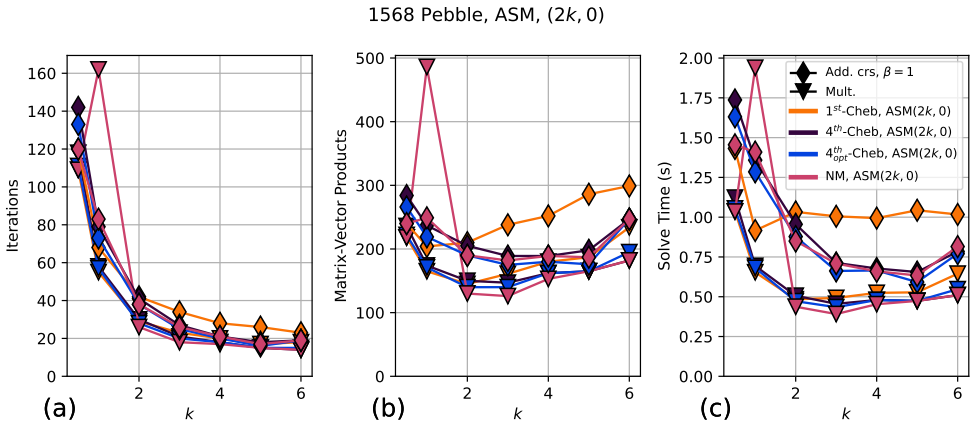



Figure B.10: 1568 pebble case (fig. 2.11b). Comparison of additive coarse-grid solve with $\beta = 1$ and multiplicative V-cycle. 12 Summit nodes ($P = 72$ V100 GPUs) is used with $n/P \sim 2.5M$. The polynomial smoother order, $k$, is varied. $(2k, 0)$ smoothing is used.

Figures B.8 to B.10 show solver performance comparisons of the additive multigrid approach with $\beta = 1$ and the standard, multiplicative V-cycle.

## B.2 LOW-ORDER AUXILIARY SPACE METHOD

The results from sections 7.2.1 and 7.2.2 are expanded here in appendices B.2.1 and B.2.2, respectively. In addition the the $(k, k)$ multigrid scheme (see section 4.1), the $(2k, 0)$ scheme is also considered. Finally, to observe the effect of tuning the $\omega_c$ parameter introduced in chapter 6, experiments with $\omega_c = 1$ are also included.

### B.2.1 Kershaw

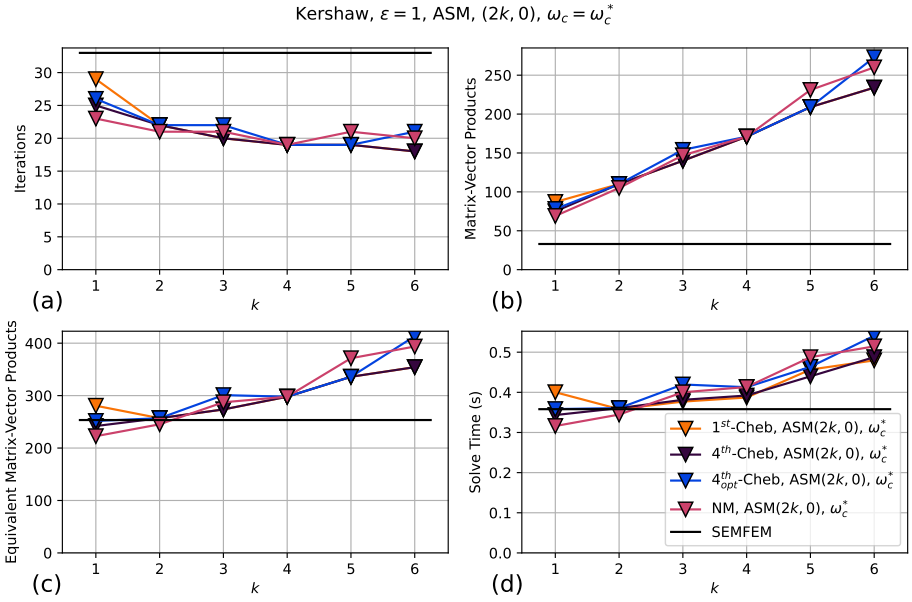



Figure B.11: Kershaw case from fig. 2.10, $\varepsilon = 1$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing. Post-smoothing is omitted, denoted by $(2k, 0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

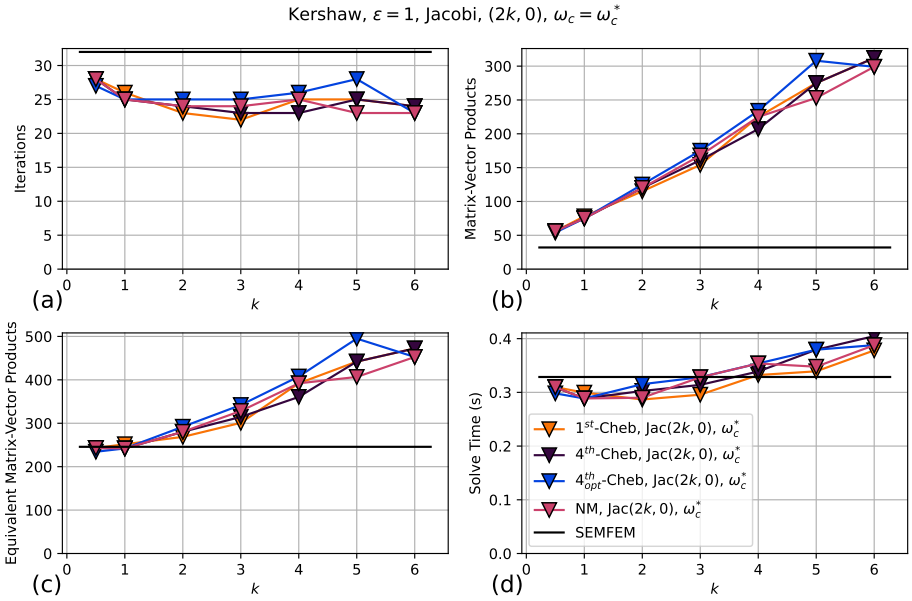



Figure B.12: Kershaw case from fig. 2.10, $\varepsilon = 1$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. Post-smoothing is omitted, denoted by $(2k, 0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

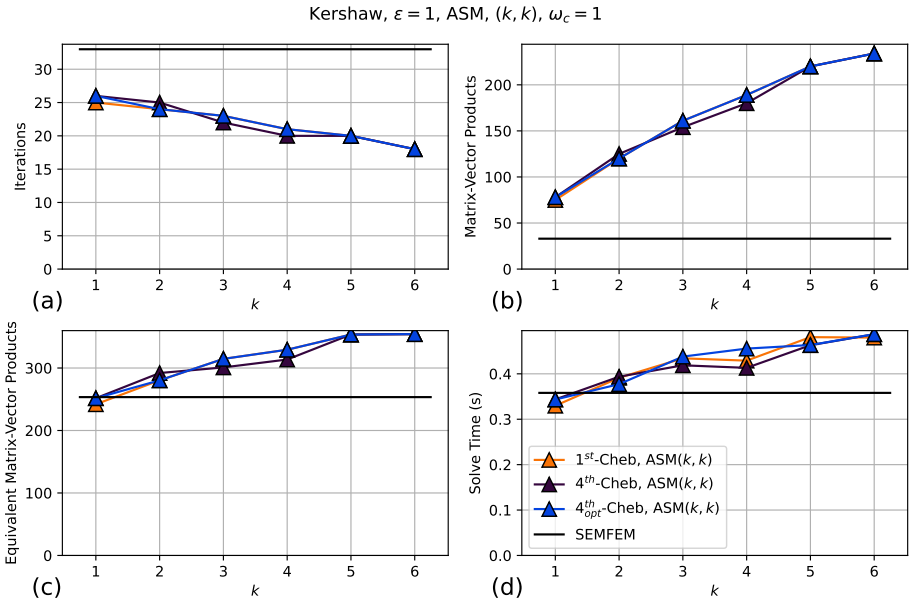



Figure B.13: Kershaw case from fig. 2.10, $\varepsilon = 1$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing. Symmetric pre-/post-smoothing is used, denoted by $(k, k)$ from chapter 4. The polynomial smoother order, $k$, is varied.

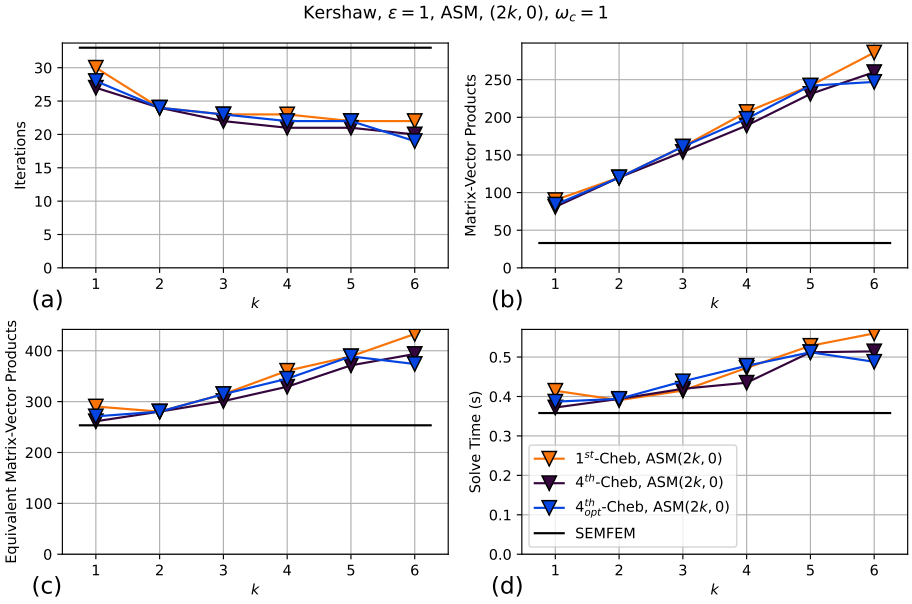



Figure B.14: Kershaw case from fig. 2.10, $\varepsilon = 1$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing. Post-smoothing is omitted, denoted by $(2k, 0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

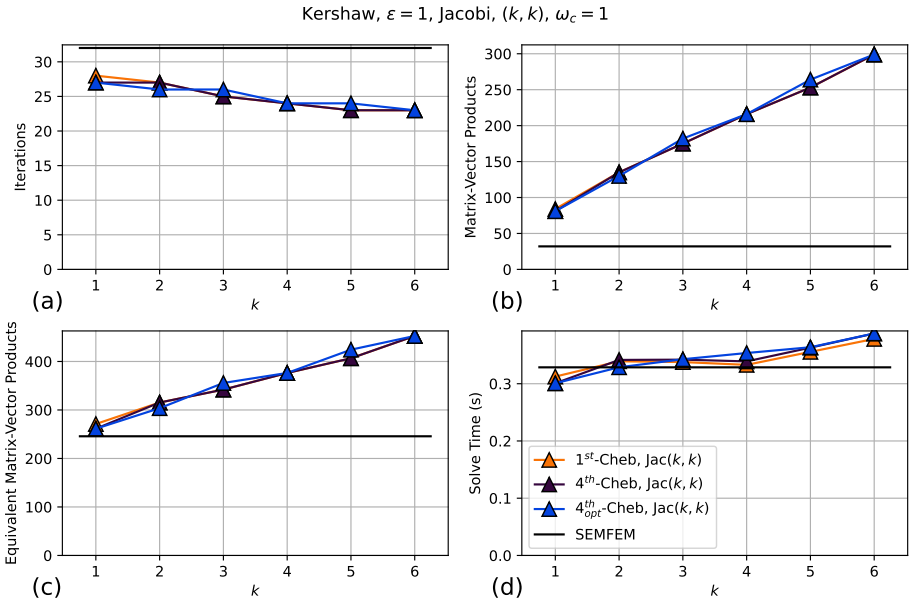



Figure B.15: Kershaw case from fig. 2.10, $\varepsilon = 1$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. Symmetric pre-/post-smoothing is used, denoted by $(k, k)$ from chapter 4. The polynomial smoother order, $k$, is varied.

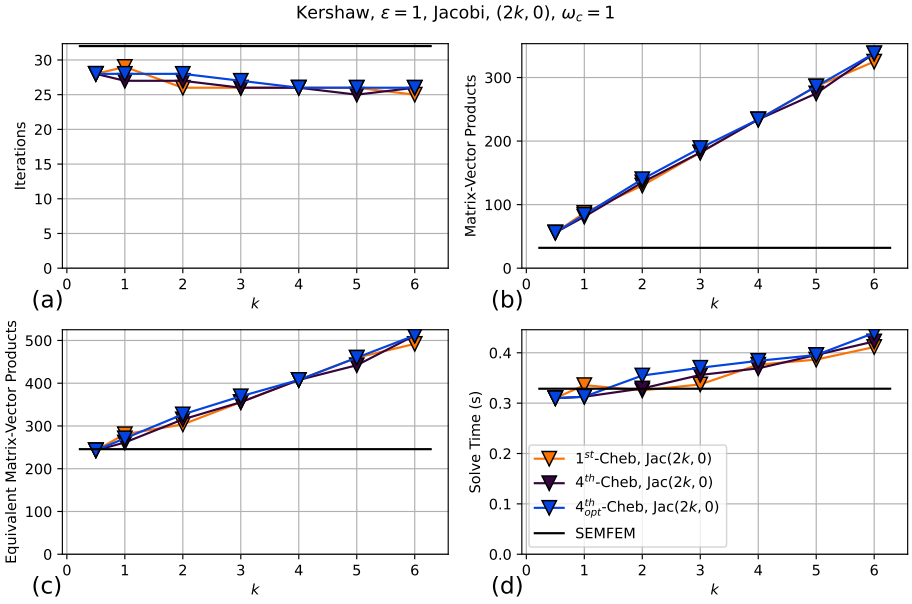



Figure B.16: Kershaw case from fig. 2.10, $\varepsilon = 1$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. Post-smoothing is omitted, denoted by $(2k, 0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

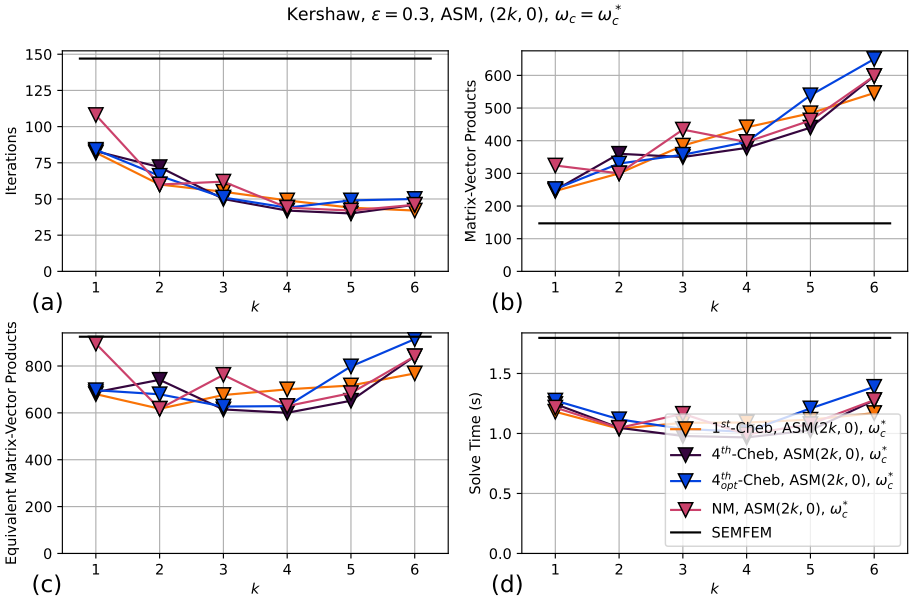



Figure B.17: Kershaw case from fig. 2.10, $\varepsilon = 0.3$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing. Post-smoothing is omitted, denoted by $(2k, 0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

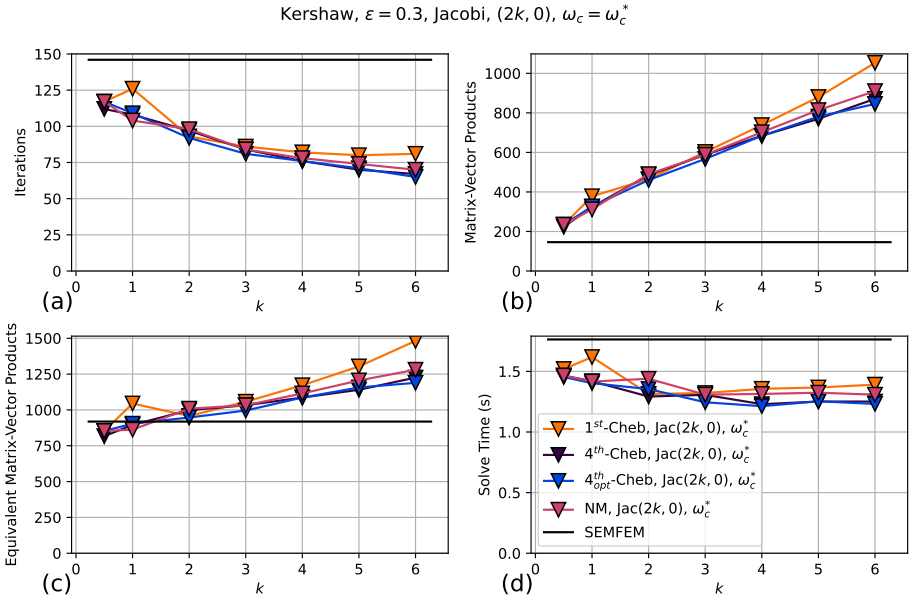



Figure B.18: Kershaw case from fig. 2.10, $\varepsilon = 0.3$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. Post-smoothing is omitted, denoted by $(2k, 0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

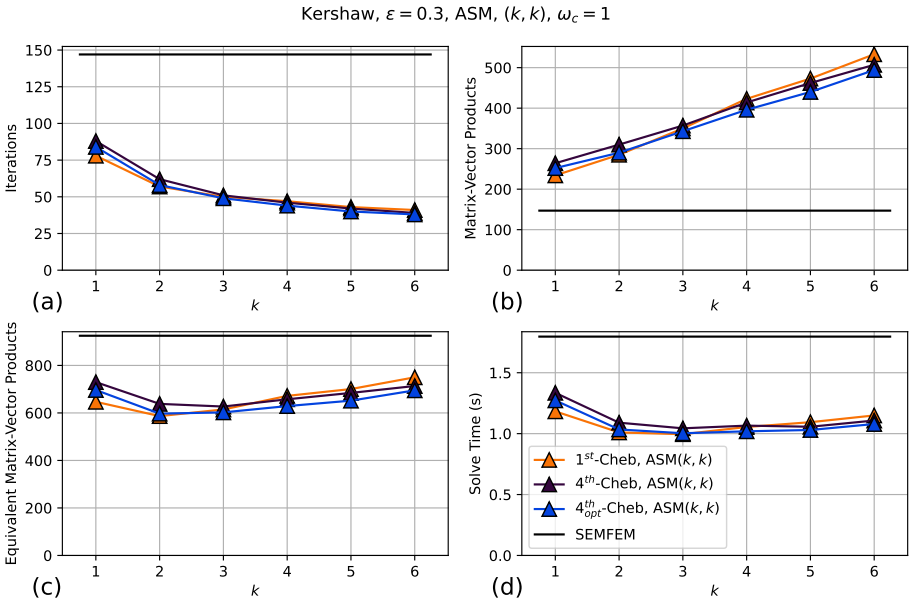



Figure B.19: Kershaw case from fig. 2.10, $\varepsilon = 0.3$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing. Symmetric pre-/post-smoothing is used, denoted by $(k, k)$ from chapter 4. The polynomial smoother order, $k$, is varied.

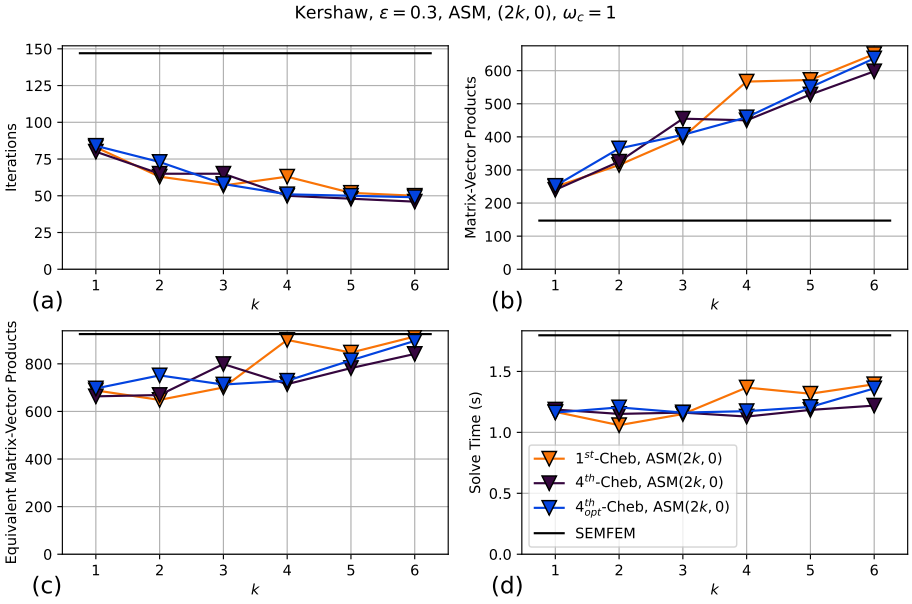



Figure B.20: Kershaw case from fig. 2.10, $\varepsilon = 0.3$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing. Post-smoothing is omitted, denoted by $(2k, 0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

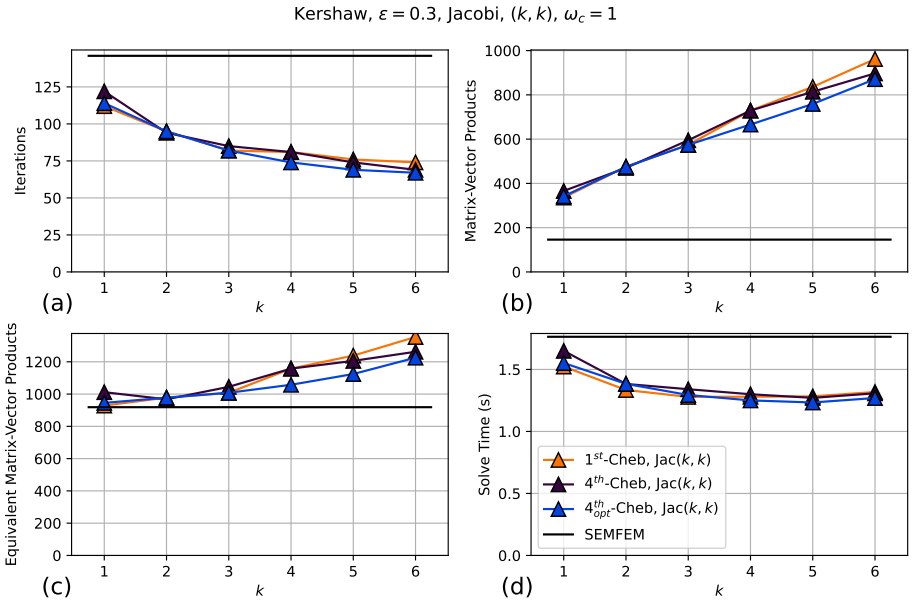



Figure B.21: Kershaw case from fig. 2.10, $\varepsilon = 0.3$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. Symmetric pre-/post-smoothing is used, denoted by $(k, k)$ from chapter 4. The polynomial smoother order, $k$, is varied.

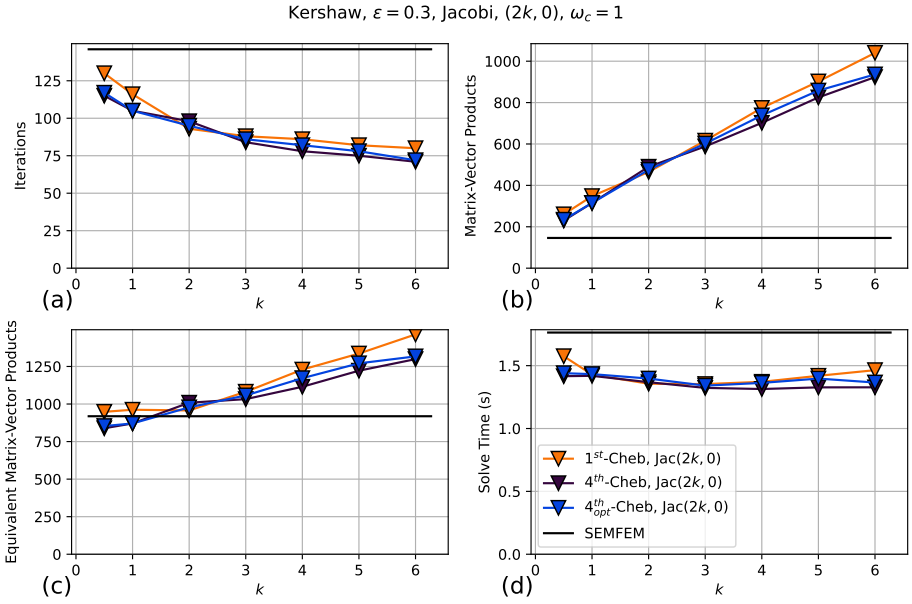



Figure B.22: Kershaw case from fig. 2.10, $\varepsilon = 0.3$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. Post-smoothing is omitted, denoted by $(2k, 0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

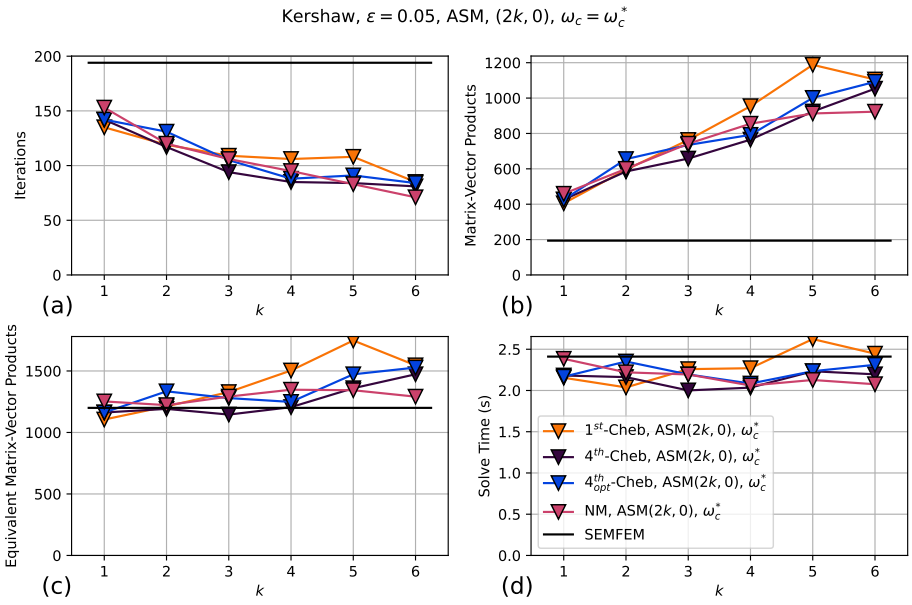



Figure B.23: Kershaw case from fig. 2.10, $\varepsilon = 0.05$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing. Post-smoothing is omitted, denoted by $(2k, 0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

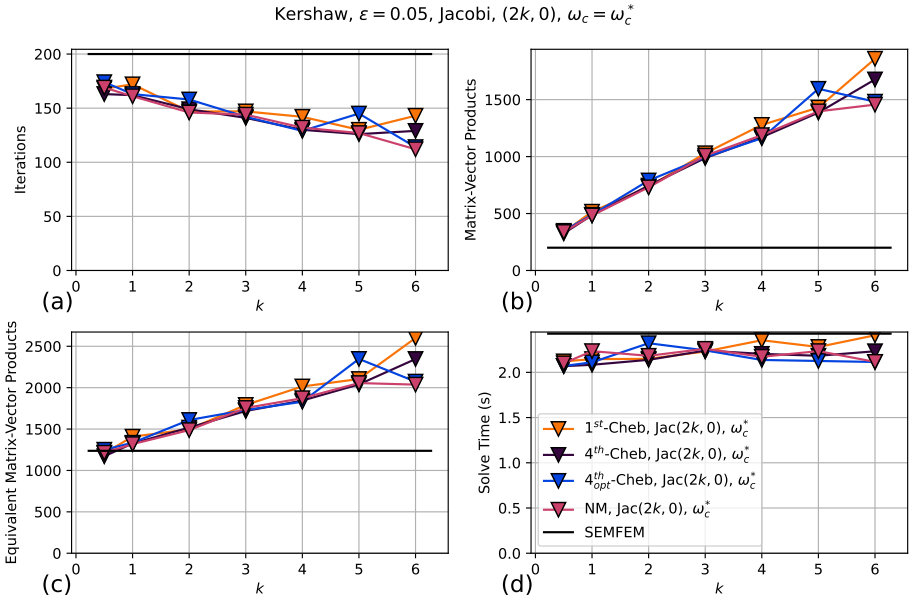



Figure B.24: Kershaw case from fig. 2.10, $\varepsilon = 0.05$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. Post-smoothing is omitted, denoted by $(2k, 0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

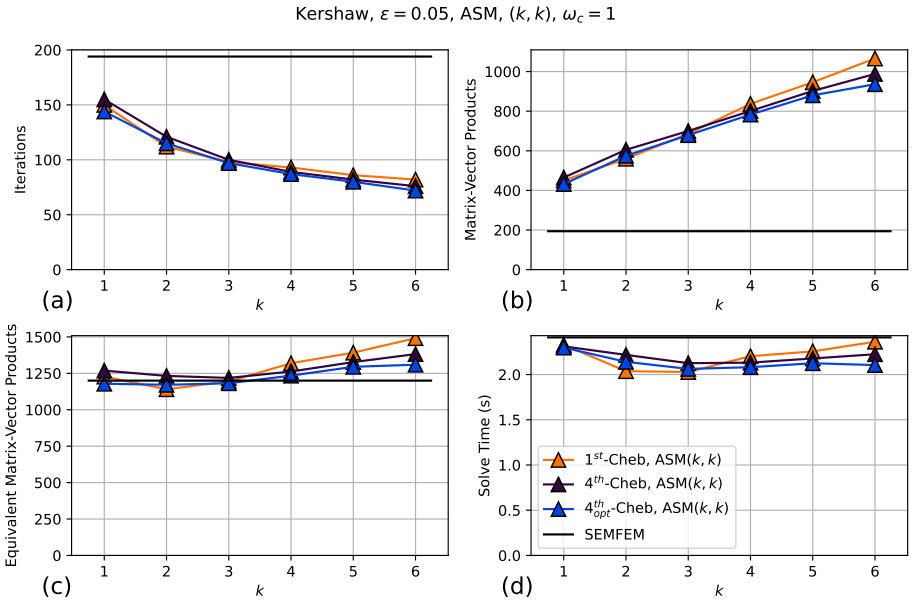



Figure B.25: Kershaw case from fig. 2.10, $\varepsilon = 0.05$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing. Symmetric pre-/post-smoothing is used, denoted by $(k,k)$ from chapter 4. The polynomial smoother order, $k$, is varied.

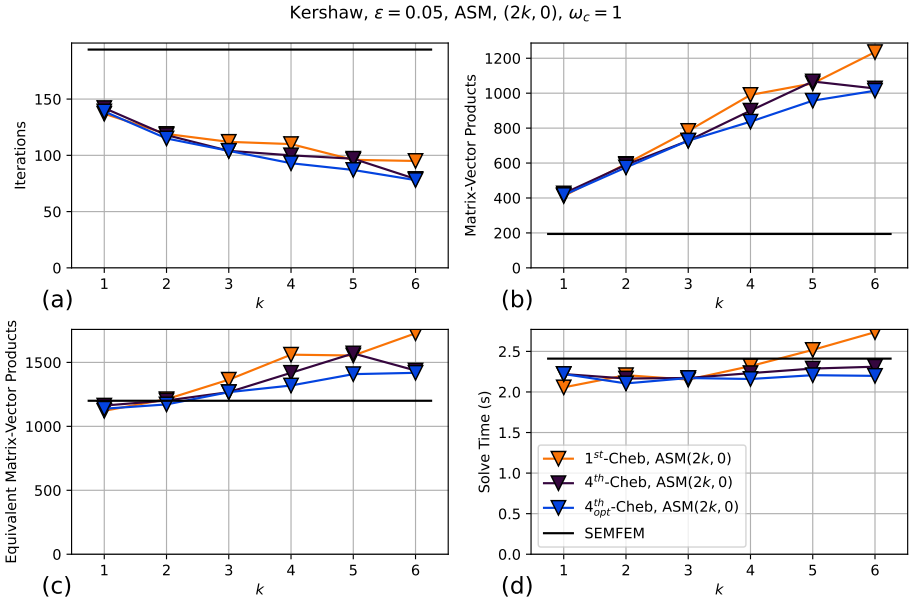



Figure B.26: Kershaw case from fig. 2.10, $\varepsilon = 0.05$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing. Post-smoothing is omitted, denoted by $(2k,0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

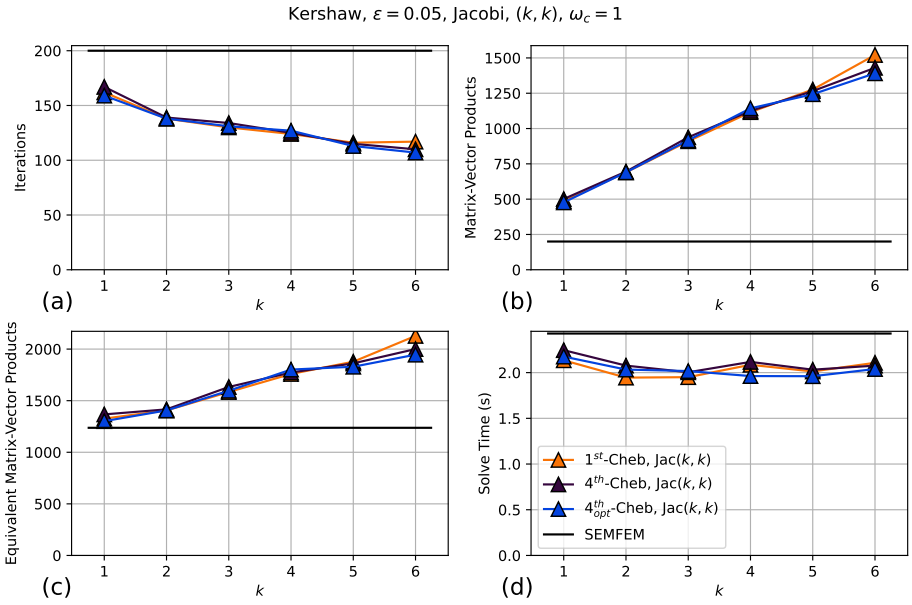



Figure B.27: Kershaw case from fig. 2.10, $\varepsilon = 0.05$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. Symmetric pre-/post-smoothing is used, denoted by $(k, k)$ from chapter 4. The polynomial smoother order, $k$, is varied.

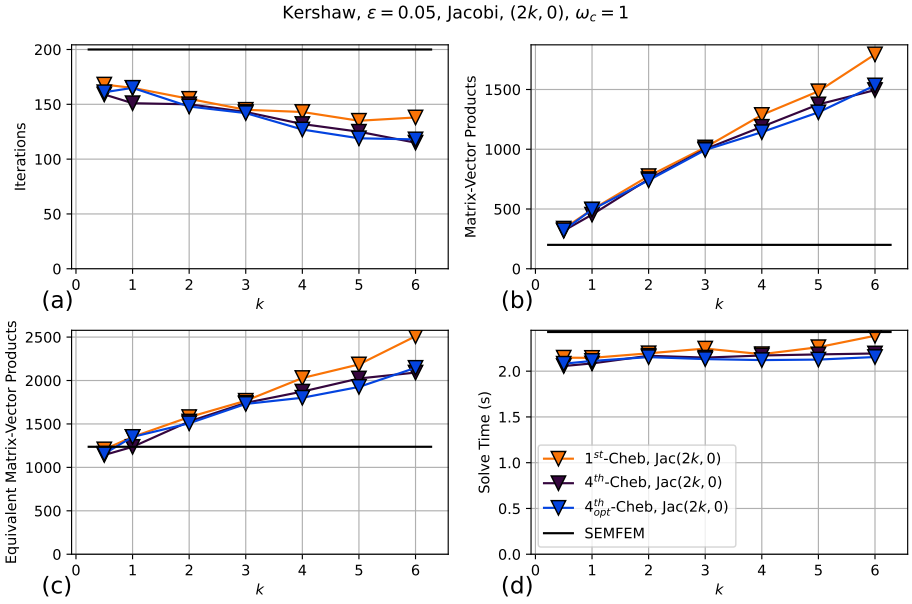



Figure B.28: Kershaw case from fig. 2.10, $\varepsilon = 0.05$. $E = 36^3, p = 7$. A single node of Summit ($P = 6$ V100 GPUs) is used with $n/P \sim 2.67M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. Post-smoothing is omitted, denoted by $(2k, 0)$ from chapter 4. The polynomial smoother order, $k$, is varied.

### B.2.2 Pebble Bed Cases

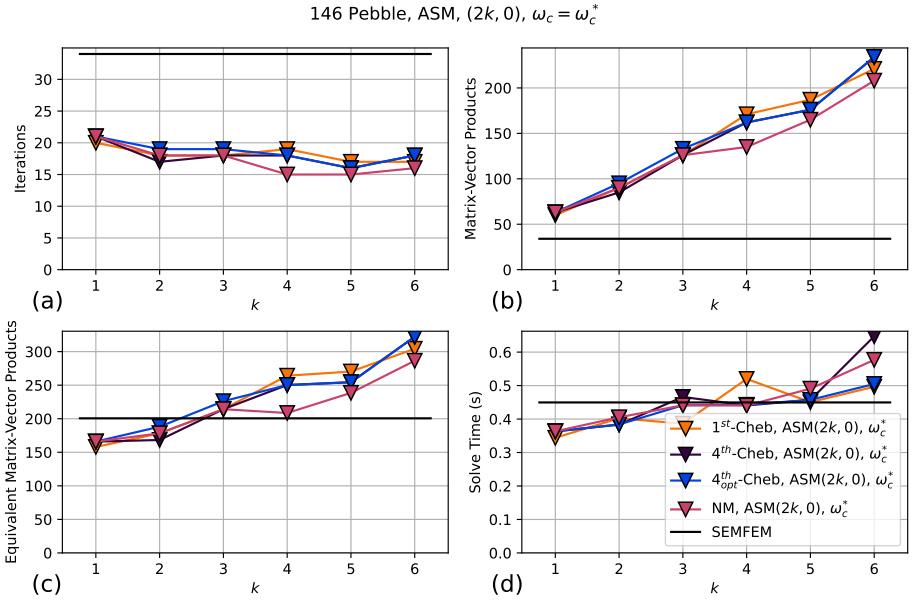



Figure B.29: 146 pebble case from fig. 2.11a. One node of Summit ($P = 6$ V100 GPUs) is used, $n/P \sim 3.5M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The $(2k, 0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

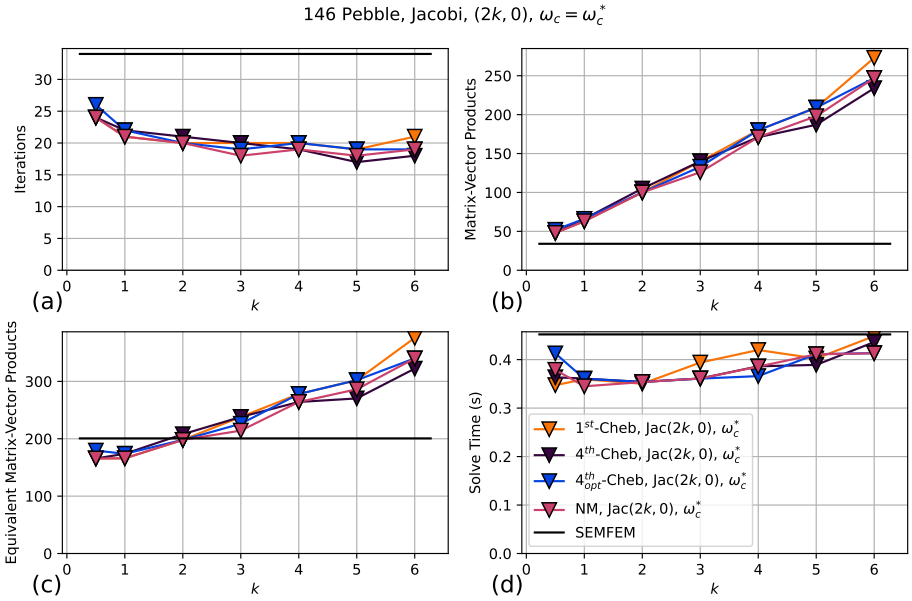



Figure B.30: 146 pebble case from fig. 2.11a. One node of Summit ($P = 6$ V100 GPUs) is used, $n/P \sim 3.5M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The $(2k, 0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

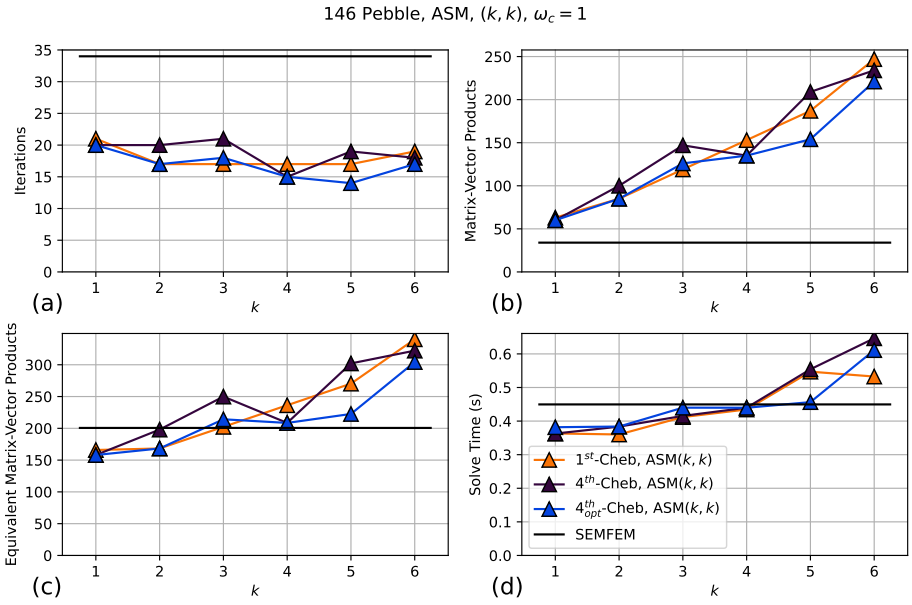



Figure B.31: 146 pebble case from fig. 2.11a. One node of Summit ($P = 6$ V100 GPUs) is used, $n/P \sim 3.5M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The $(k,k)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

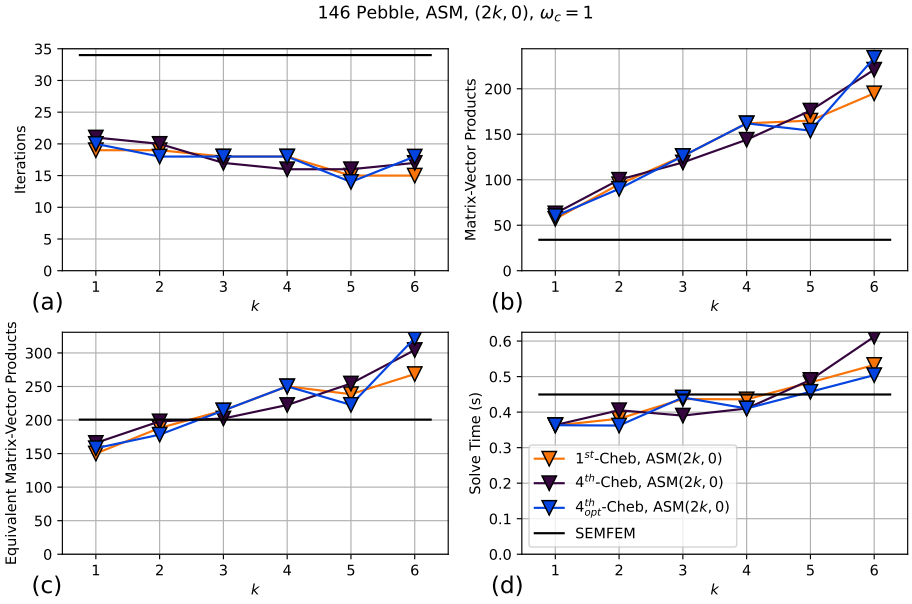



Figure B.32: 146 pebble case from fig. 2.11a. One node of Summit ($P = 6$ V100 GPUs) is used, $n/P \sim 3.5M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The $(2k,0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

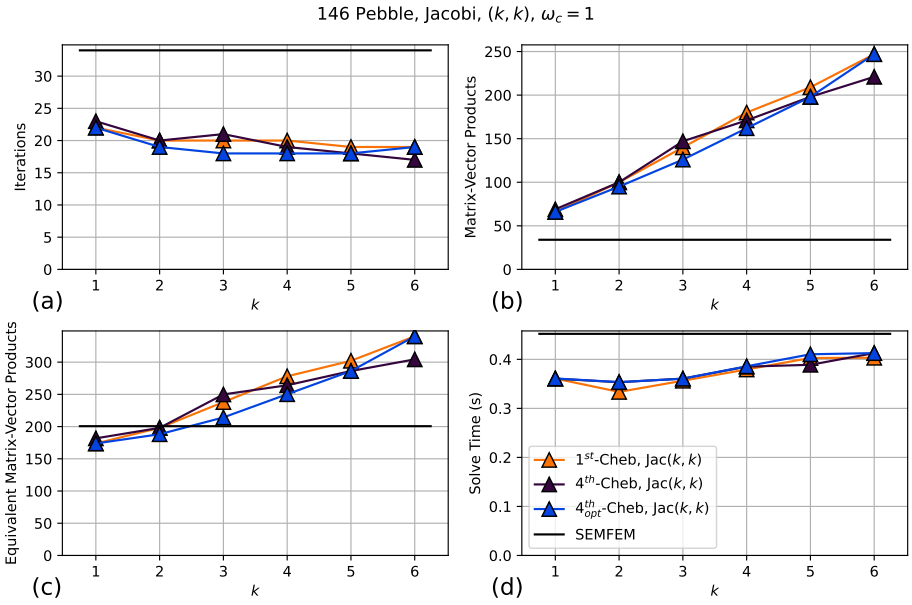



Figure B.33: 146 pebble case from fig. 2.11a. One node of Summit ($P = 6$ V100 GPUs) is used, $n/P \sim 3.5M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The $(k,k)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

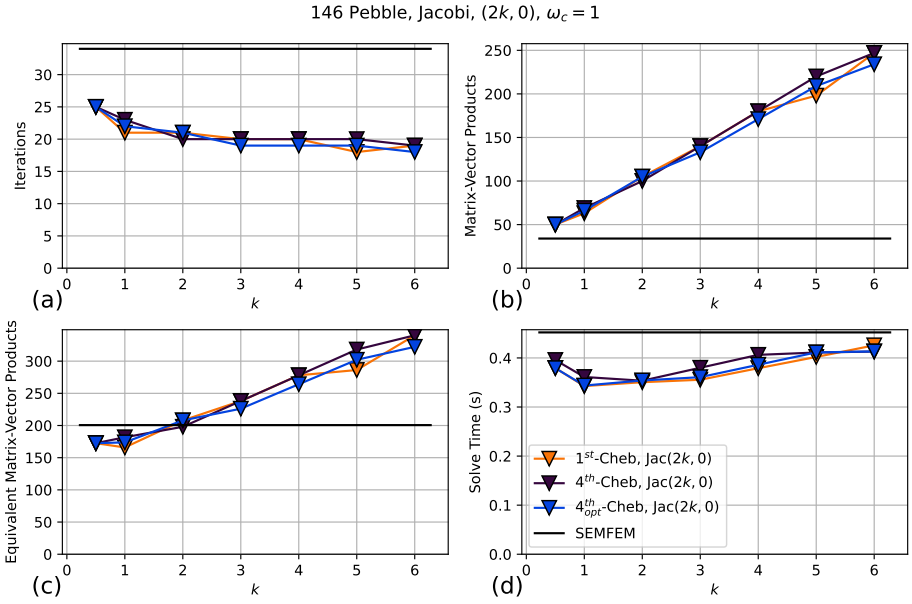



Figure B.34: 146 pebble case from fig. 2.11a. One node of Summit ($P = 6$ V100 GPUs) is used, $n/P \sim 3.5M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi. The $(2k,0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

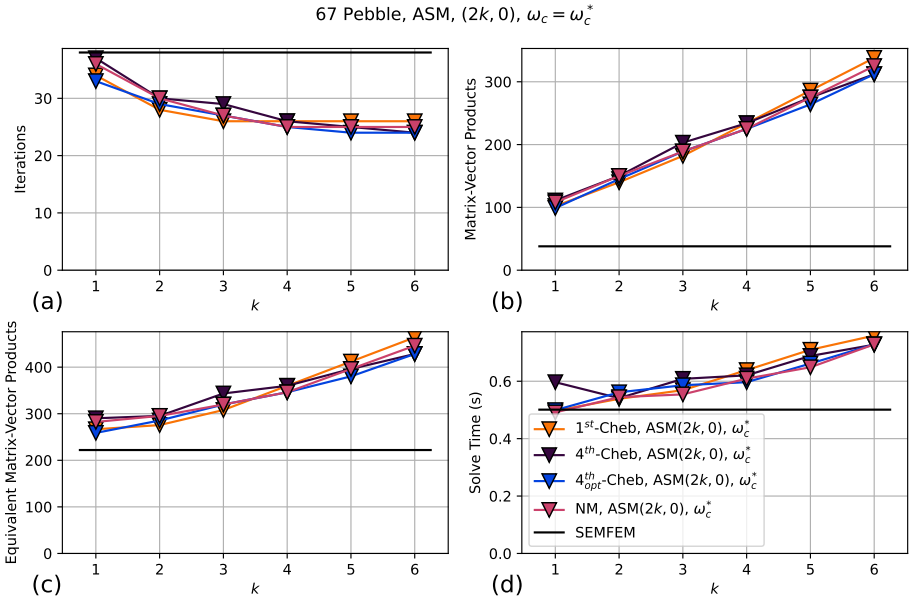



Figure B.35: 67 pebble case from fig. 2.11c. Three nodes of Summit ($P = 18$ V100 GPUs) is used, $n/P \sim 2.33M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The $(2k, 0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

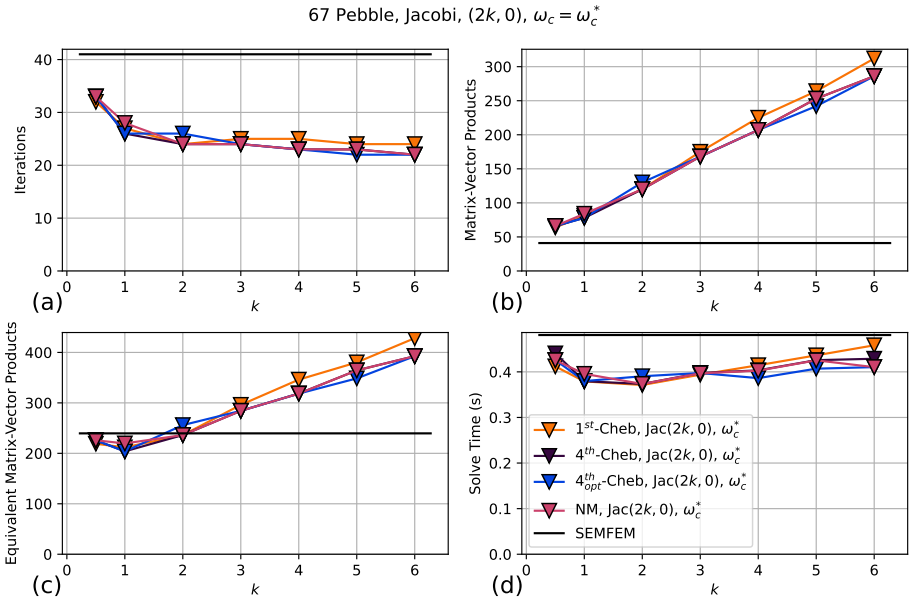



Figure B.36: 67 pebble case from fig. 2.11c. Three nodes of Summit ($P = 18$ V100 GPUs) is used, $n/P \sim 2.33M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The $(2k, 0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

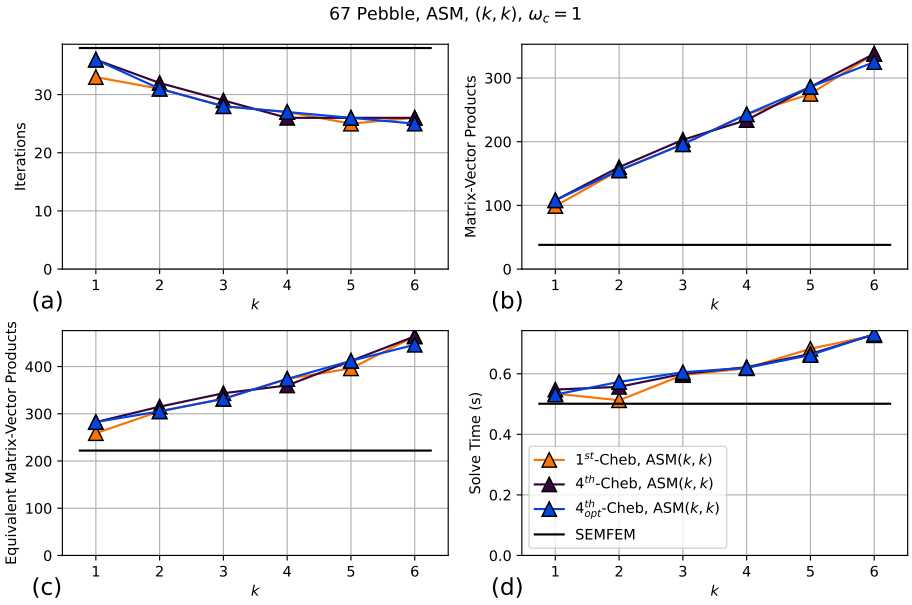



Figure B.37: 67 pebble case from fig. 2.11c. Three nodes of Summit ($P = 18$ V100 GPUs) is used, $n/P \sim 2.33M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The $(k,k)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

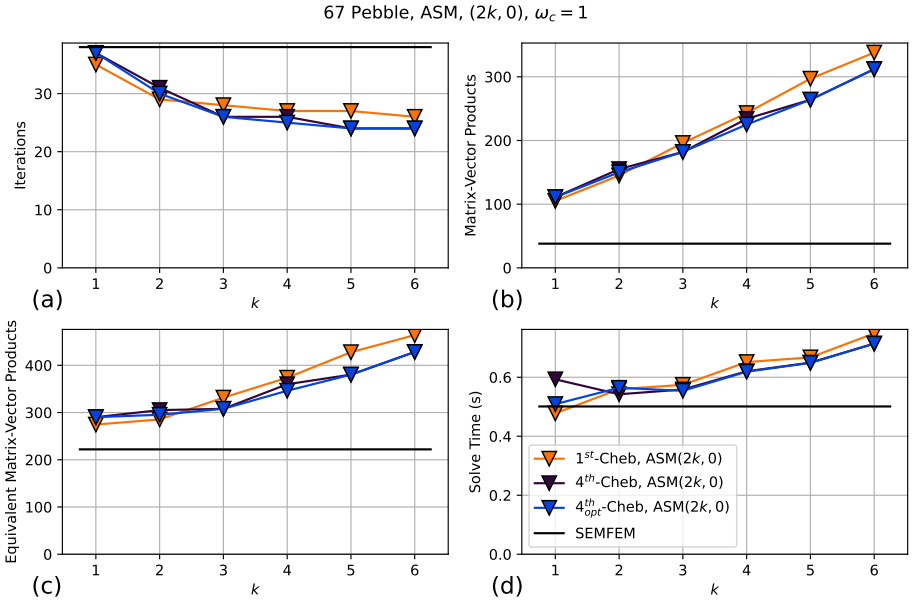



Figure B.38: 67 pebble case from fig. 2.11c. Three nodes of Summit ($P = 18$ V100 GPUs) is used, $n/P \sim 2.33M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The $(2k,0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

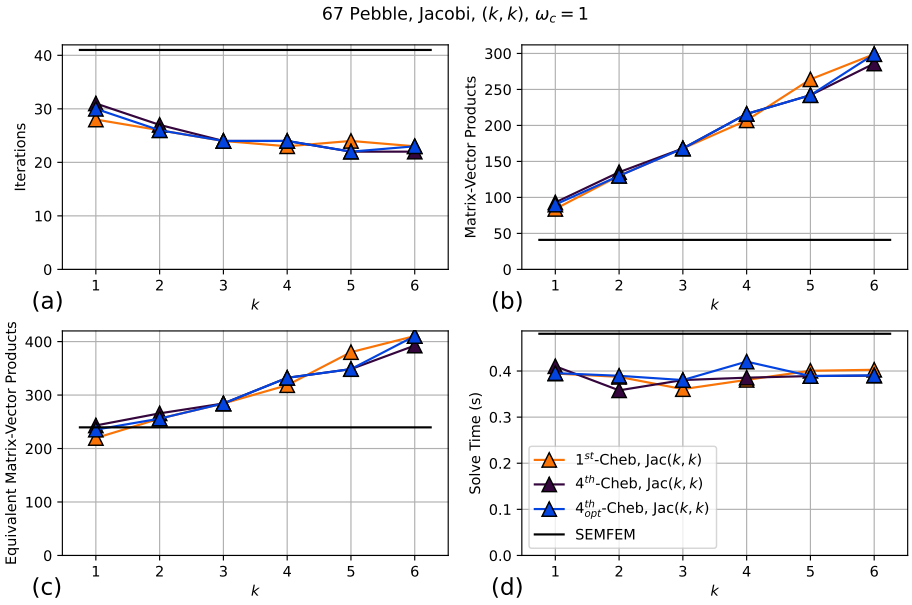



Figure B.39: 67 pebble case from fig. 2.11c. Three nodes of Summit ($P = 18$ V100 GPUs) is used, $n/P \sim 2.33M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The $(k,k)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

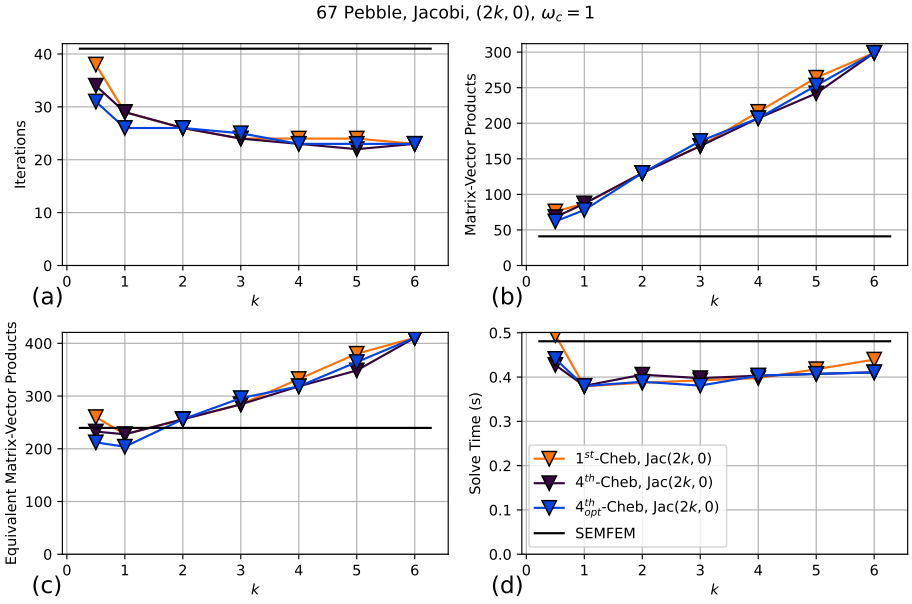



Figure B.40: 67 pebble case from fig. 2.11c. Three nodes of Summit ($P = 18$ V100 GPUs) is used, $n/P \sim 2.33M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi. The $(2k,0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

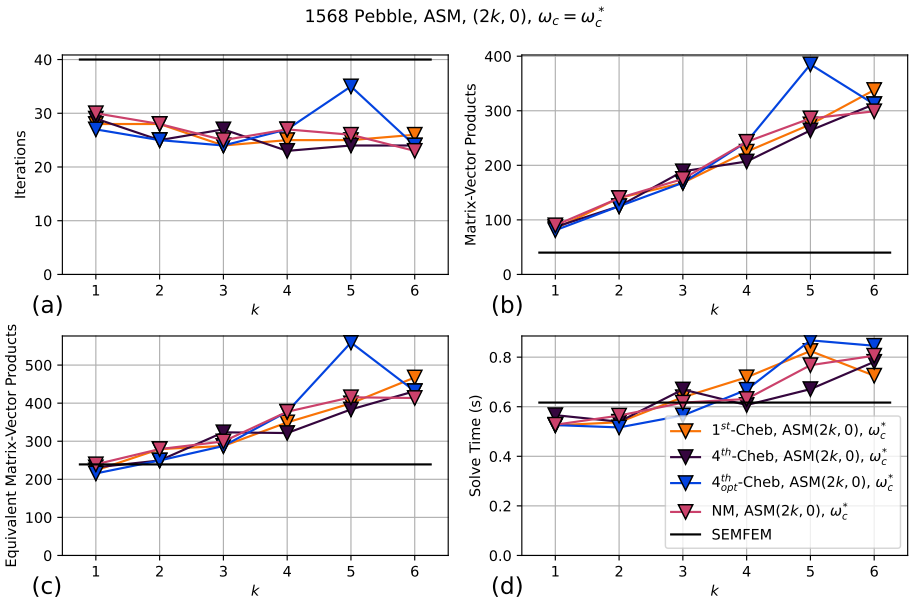



Figure B.41: 1568 pebble case from fig. 2.11b. 12 nodes of Summit ($P = 72$ V100 GPUs) is used, $n/P \sim 2.5M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The $(2k, 0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

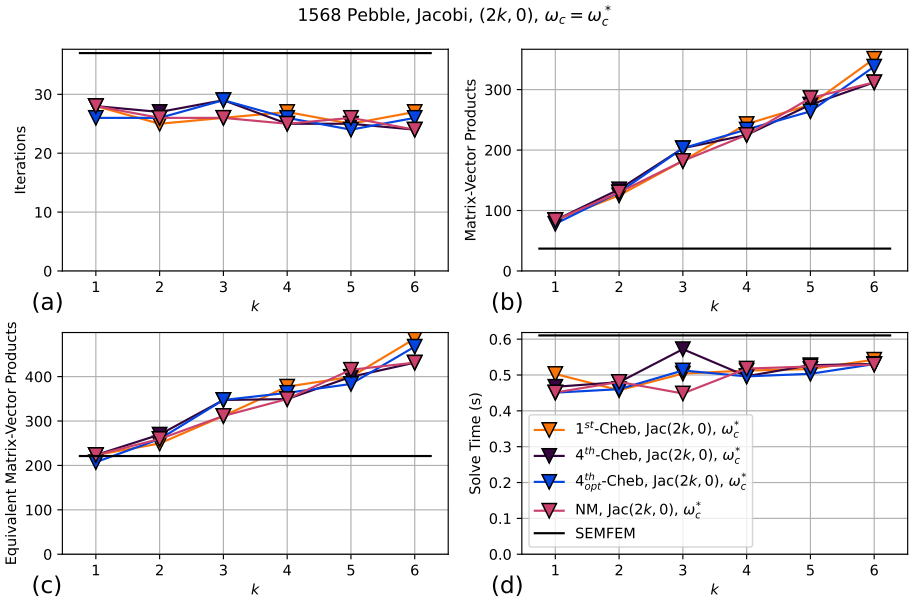



Figure B.42: 1568 pebble case from fig. 2.11b. 12 nodes of Summit ($P = 72$ V100 GPUs) is used, $n/P \sim 2.5M$. HOS-SEMFEM preconditioner with tuned $\omega_c$ value from chapter 6. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The $(2k, 0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

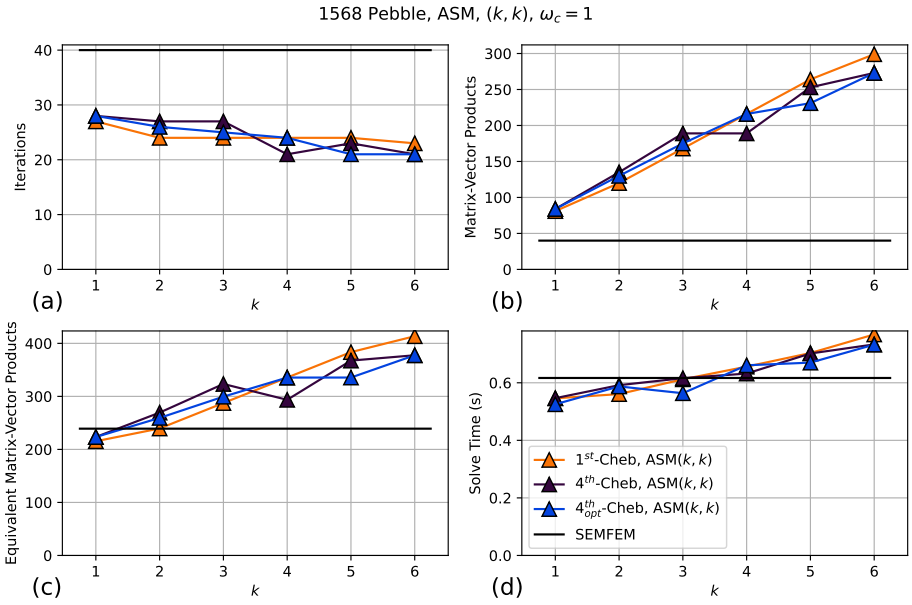



Figure B.43: 1568 pebble case from fig. 2.11b. 12 nodes of Summit ($P = 72$ V100 GPUs) is used, $n/P \sim 2.5M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The $(k,k)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

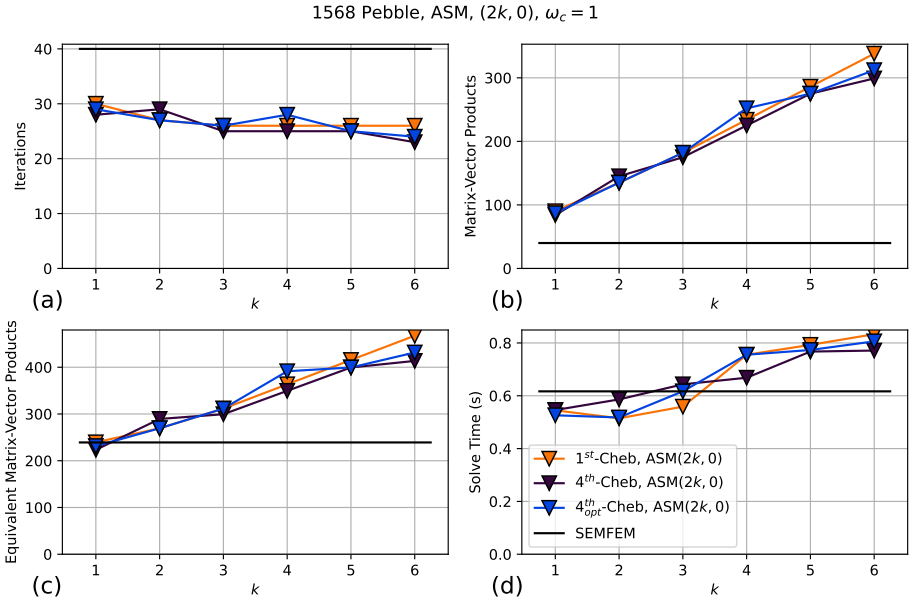



Figure B.44: 1568 pebble case from fig. 2.11b. 12 nodes of Summit ($P = 72$ V100 GPUs) is used, $n/P \sim 2.5M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with ASM smoothing from alg. 2.10. The $(2k,0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

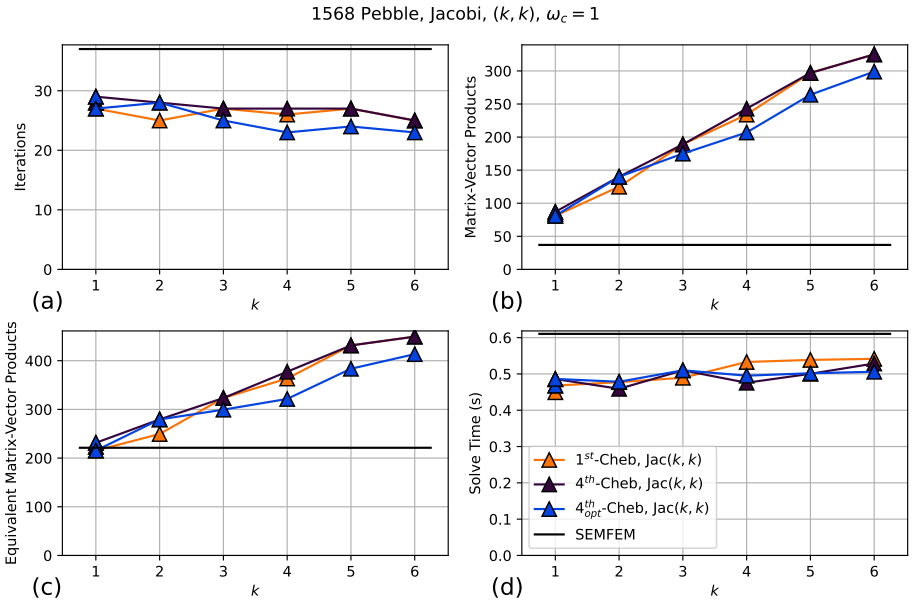



Figure B.45: 1568 pebble case from fig. 2.11b. 12 nodes of Summit ($P = 72$ V100 GPUs) is used, $n/P \sim 2.5M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi smoothing. The $(k,k)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.

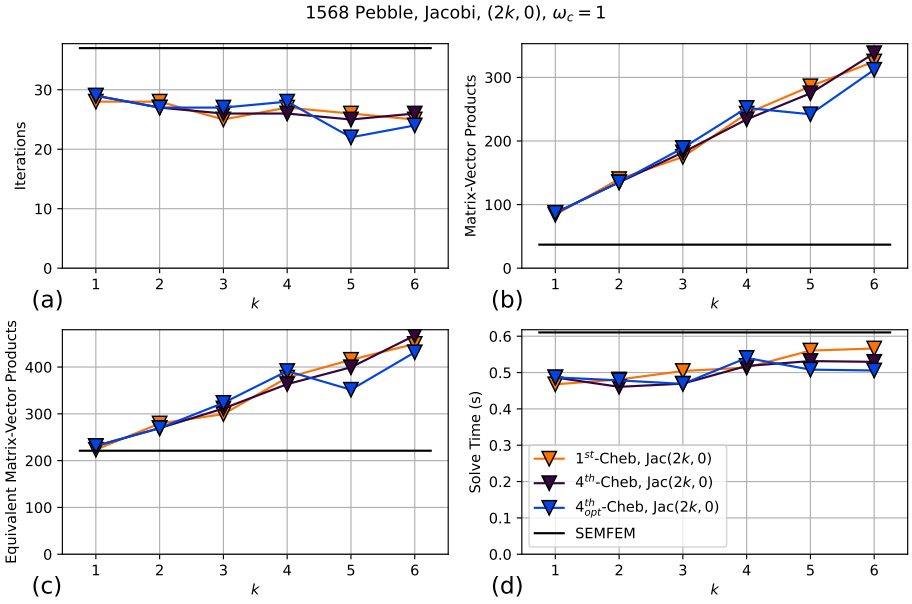



Figure B.46: 1568 pebble case from fig. 2.11b. 12 nodes of Summit ($P = 72$ V100 GPUs) is used, $n/P \sim 2.5M$. HOS-SEMFEM preconditioner with $\omega_c = 1$. The polynomial smoothers introduced in sections 2.7 and 4.1 are used with Jacobi. The $(2k,0)$ cycle scheme from chapter 4 is used. The polynomial smoother order, $k$, is varied.