`xhdfe`: Fast high-dimensional fixed-effects estimation in Stata

Miguel Portela*
NIPE, University of Minho, BPLIM, Banco de Portugal, and IZA@LISER Network

Tiago Tavares†
NIPE, University of Minho

August 2026

Abstract

`xhdfe` is a Stata command for linear regression with high-dimensional fixed effects. It targets the same partialled-out least-squares estimator as `reghdfe` on the overlapping specifications studied here, while moving the main absorption workload into a compiled C++ backend with multithreaded CPU execution and optional NVIDIA CUDA acceleration. The command preserves a Stata e-class interface while supporting multiple absorbed fixed effects, clustered and robust standard errors, heterogeneous slopes in the tested specifications, backend choices, saved fixed effects, residuals, fitted values, and stored results. We validate `xhdfe` on two large wage regressions using Portuguese matched employer-employee data. The command reproduces `reghdfe` coefficients and standard errors within tight numerical tolerances and delivers speedups above 100-fold with CPU execution and 200-fold with CUDA acceleration in the fastest configurations. Compared with alternative high-dimensional fixed-effects implementations, `xhdfe` records the shortest runtimes among the implementations and configurations reported here. The development history also provides a case study of AI-assisted software development. Agentic tools supported coding and review, while conventional numerical tests and reproducible benchmarks assessed the resulting changes.

Keywords: `xhdfe`, `reghdfe`, high-dimensional fixed effects, Stata, worker-firm data, GPU acceleration, CUDA, AI-assisted software development

*Corresponding author. NIPE, School of Economics, Management and Political Science, University of Minho, Campus de Gualtar, 4710-057 Braga, Portugal. He is also affiliated with BPLIM, Banco de Portugal, and the IZA@LISER Network. Email: miguel.portela@eeg.uminho.pt.

†NIPE, School of Economics, Management and Political Science, University of Minho. Email: tgstavares@eeg.uminho.pt.

1 Introduction

High-dimensional fixed effects (HDFE) are now a standard part of applied empirical work. Labor economists absorb worker, firm, occupation, job-title, and time effects. Trade and industrial-organization applications absorb product, market, location, or firm-by-time effects. Public-finance and finance applications often combine several of these dimensions. The canonical worker-firm setting of Abowd et al. (1999) illustrates the statistical appeal: fixed effects allow researchers to separate worker and firm heterogeneity in large matched datasets. The same appeal makes computation a central practical constraint. As shown by Guimarães and Portugal (2010) and by applications such as Carneiro et al. (2012), the econometric problem is often not a small regression with a few nuisance controls, but a large repeated residualization problem with millions of observations and many absorbed categories.

Stata users already have mature tools for these models, especially `reghdfe`[1] (Correia, 2017, Correia and Constantine, 2014), together with newer official high-dimensional fixed-effects functionality in Stata (StataCorp, 2026). These tools made HDFE estimation routine in applied work. For the largest applications, however, runtime and memory constraints make repeated estimation costly. A specification that takes minutes or hours to estimate changes how researchers search over models, test robustness, bootstrap, compare inference choices, or reproduce a complete empirical analysis.

This article introduces `xhdfe`, a Stata command for linear regression with high-dimensional fixed effects.[2] The objective is deliberately narrow: `xhdfe` is designed to compute the same linear HDFE estimand as `reghdfe` on overlapping specifications, while moving the main numerical bottleneck into a compiled backend. Users call `xhdfe` through ordinary Stata syntax, receive Stata e-class results, and can use familiar options such as `absorb()`, `vce()`, `cluster()`, `residuals()`, `keepsingletons`, and `numthreads()`. The heavy absorption and regression work is handled by C++ code exposed through a Stata plugin, with optional GPU execution when a GPU-enabled plugin and device are available. Throughout the paper, GPU refers to accelerator execution in general. We use CUDA when reporting the backend actually run and the benchmark results, and GPU when the statement is generic to non-CPU execution.

The paper makes four contributions. First, the development of `xhdfe` provides a case study in AI-assisted improvement of mature empirical software. Agentic tools supported implementation, refactoring, and code review under the authors' supervision. Econometric validation remained independent of those tools, with each change required to preserve the `reghdfe` estimand and reported results.

Second, the paper documents the estimator target, algorithmic choices, software archi-

[1]The `reghdfe` software repository is available at `https://github.com/sergiocorreia/reghdfe`.

[2]The `xhdfe` package (ado-file, help files, compiled plugin, C++/CUDA sources, and replication do-files) is available from its public repository at `https://github.com/reisportela/xhdfe-xfe`. The command requires Stata 16 or later. The repository README documents the supported operating systems, the prebuilt CPU plugins, and how to rebuild the plugin, including the CUDA build, from source. The repository also ships Python and R packages that expose the same compiled core outside Stata. This article documents the Stata command.

tecture, command syntax, and stored results of `xhdfe`. The command is not a complete replacement for every `reghdfe` feature. We document and validate a substantial subset of linear HDFE estimation used in applied work: multiple categorical fixed effects, robust and clustered inference (including multiway clustering), singleton handling, categorical fixed-effect interactions, factor-variable slope syntax in tested cases, fixed-effect contribution saving, and selected postestimation features. The package additionally ships companion commands built on the same compiled core and documented in Section 4.9, including a standalone partialling-out command, `xfe`, and worker-firm tools for leave-out variance decompositions with Kline et al. (2020) bias correction and Gelbach (2016) coefficient decompositions.

Third, the paper provides validation and runtime evidence on two large wage regressions in the Abowd, Kramarz and Margolis (AKM) style, using Portuguese matched employer-employee data. The agreement results show that `xhdfe` reproduces `reghdfe` coefficients and standard errors within tight numerical tolerances. The runtime gains are large. Under the current default numerical criteria, the Stata `xhdfe` command is 41.0 times faster than `reghdfe` on CPU and 141.4 times faster on CUDA in the common-seniority specification. In the specification with heterogeneous fixed-effect slopes, the corresponding speedups are 71.2 and 206.3 times. The Python wrapper, which calls the same compiled core, reaches speedups of 209.9 and 236.4 times on CUDA under the same criteria. With a faster stopping criterion, the largest reported speedups are 252.2 and 287.2 times.

Fourth, the paper compares `xhdfe` with high-dimensional fixed-effects implementations in R, Python, and Julia. The comparison reports runtime, numerical agreement with `reghdfe`, backend, and support for heterogeneous slopes. The closest non-`xhdfe` runtime comes from `FixedEffectModels.jl` on CUDA, while `pyfixest` is the fastest CPU-only non-`xhdfe` implementation in the simpler specification. In the benchmarked version, `pyfixest` does not support the heterogeneous-slope specification used in the second benchmark.

The remainder of the article proceeds as follows. Section 2 describes the development process and the role of AI assisted tools. Section 3 describes the estimator target, software architecture, and computational mechanisms behind the runtime gains. Section 4 describes syntax, options, stored results, and companion commands, including a standalone partialling-out command and two worker-firm decomposition tools. Section 5 validates and benchmarks `xhdfe` on two large AKM-style regressions and compares it with other Stata, R, Python, and Julia implementations. The article then concludes.

2 Development strategy and history of `xhdfe`

The development of `xhdfe` followed a deliberately conservative target: accelerate established `reghdfe`-style estimation without changing the estimator or the Stata interface. `reghdfe` is widely used in applied economics and sets the practical standard for linear regressions with high-dimensional fixed effects in Stata. The development task was therefore to preserve the numerical behavior of overlapping `reghdfe` specifications while moving the computational bottleneck into compiled code designed for large repeated projection problems.

Speed improvements mattered only after numerical agreement was established. Numerical

validation first required agreement with the relevant `reghdfe` outputs: coefficients, standard errors, estimation sample sizes, singleton handling, degrees of freedom, and selected stored results. Once those checks defined the target, optimization focused on moving the within-transformation and covariance calculations into C++, adding CPU threading, exposing backend metadata, and testing optional CUDA execution.

Agentic AI tools assisted with coding, refactoring, and review under the authors' direction. The authors specified the objectives, reviewed the proposed changes, and decided what entered the code base. Early AI-assisted work helped establish the command interface and connect the Stata ado layer to compiled code. Later work focused on diagnosing numerical discrepancies, improving the C++ absorption routines, preserving Stata e-class output, and explaining the sources of the runtime gains. Larger models available through Codex and Claude Code were also used for broader refactoring and code review. Changes were kept only when the numerical tests continued to pass. The repository makes these checks public through a Stata certification suite and a benchmark replication suite.[3]

The historical development benchmark quantifies the runtime implications of those changes. It compares successive development versions that introduced performance improvements, GPU support, numerical corrections, or AI-assisted revisions. Each available CPU or CUDA implementation is measured with ten repetitions. The benchmark uses a restricted Quadros de Pessoal (QP) matched worker-firm file with 46,156,187 estimation observations after singleton handling.[4] The absorbed dimensions contain 5,074,679 worker categories, 610,089 firm categories, and 26 year categories. The model regresses log real hourly wage on age squared, tenure, tenure squared, and education, absorbing worker, firm, and year fixed effects and clustering standard errors by worker. The `reghdfe` reference time for this specification is 1,379 seconds. This historical development benchmark is separate from the full AKM benchmark reported in Section 5. It uses an earlier restricted QP extract and a single specification with common seniority coefficients to track runtime across development versions. Figure 1 shows the median execution times for the historical `xhdfe` CPU implementations.

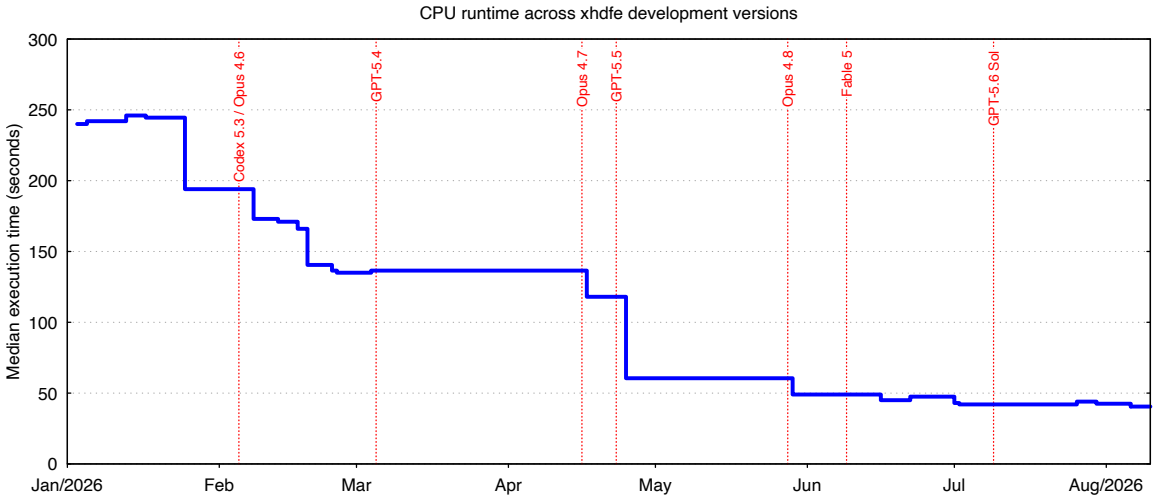
Runtime fell substantially after the estimator target had been fixed. Early CPU implementations were already faster than the `reghdfe` baseline, but still took roughly four minutes on the restricted benchmark. Later refactoring reduced the median CPU runtime to 40.5 seconds in the latest benchmarked release. The timing of individual declines cannot be attributed to particular model releases. The figure instead summarizes several months of implementation, refactoring, and review in which each optimization was checked against existing numerical results.

The same approach is not specific to high-dimensional fixed effects. Many empirical projects rely on estimators that are conceptually settled but computationally expensive in practice, including nonparametric regressions, bootstrap procedures, nonlinear VARs, particle

[3]The directory `tests/stata` contains certification tests adapted from the `reghdfe` test-suite style. Each test runs a reference Stata command (`regress`, `areg`, `reghdfe`, or `ivreghdfe`) on a small synthetic dataset and compares coefficients, standard errors, estimation samples, degrees of freedom, and stored results. The directory `tests/benchmarks` contains public replication material for the specifications used to track performance across implementations and software versions. Its Stata, Python, and R runners record timings and coefficients, allowing numerical agreement to be checked alongside runtime.

[4]We thank Paulo Guimarães for enabling us to test `xhdfe` on the data used in Portugal et al. (2026).

Figure 1: CPU runtime fell sharply across `xhdfe` development versions



Notes: For each benchmarked version with a working CPU implementation, the step function reports the median runtime across ten repetitions using 16 threads. Each value is carried forward until the next benchmarked version. Dashed vertical lines locate the benchmark history relative to contemporaneous AI model releases. They provide historical context rather than causal effects of individual model releases.

filters, and simulation-based estimators. In such cases, an established implementation provides a benchmark for the statistical results, while numerical tests reveal whether faster code changes those results. The development history of `xhdfe` shows how AI-assisted engineering can improve mature research software when numerical agreement and reproducible timing evidence remain the basis for evaluation.

3 Estimator and computational architecture

The runtime evidence in Sections 2 and 5 is informative because `xhdfe` targets the same linear fixed-effects estimand as `reghdfe` for the overlapping specifications studied here. The software contribution is not a new estimator, but a faster way to compute the within transformation that makes the estimator feasible in large repeated applications.

3.1 Estimator target

Consider the linear high-dimensional fixed-effects model

$$y_i = \mathbf{x}_i'\boldsymbol{\beta} + \sum_{g=1}^{G} \mathbf{d}_{ig}'\boldsymbol{\alpha}_g + u_i \qquad i = 1, \ldots, n$$

where $\mathbf{x}_i \in \mathbb{R}^K$ contains the slope covariates and $\mathbf{d}_{ig}$ is the row of the dummy design for fixed-effect dimension g. Let

$$\mathbf{y} \in \mathbb{R}^n \qquad \mathbf{X} \in \mathbb{R}^{n \times K} \qquad \mathbf{D} \equiv [\mathbf{D}_1, \ldots, \mathbf{D}_G]$$

For the derivation below, suppose that the optional diagonal weight matrix $\mathbf{W} \equiv \text{diag}(w_i)$ has strictly positive entries. The least-squares target is

$$(\widehat{\boldsymbol{\beta}}, \widehat{\boldsymbol{\alpha}}) \in \arg\min_{\boldsymbol{\beta},\boldsymbol{\alpha}} (\mathbf{y} - \mathbf{X}\boldsymbol{\beta} - \mathbf{D}\boldsymbol{\alpha})'\mathbf{W}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta} - \mathbf{D}\boldsymbol{\alpha})$$

The matrix $\mathbf{D}$ is too large to construct explicitly in the applications for which HDFE commands are useful. Instead, estimation uses the weighted Frisch-Waugh-Lovell (FWL) representation. Define

$$\mathbf{P}_D \equiv \mathbf{D}(\mathbf{D}'\mathbf{W}\mathbf{D})^{+}\mathbf{D}'\mathbf{W} \qquad \mathbf{M}_D \equiv \mathbf{I} - \mathbf{P}_D \tag{1}$$

where $(\cdot)^{+}$ denotes a generalized inverse. If $\widetilde{\mathbf{y}} \equiv \mathbf{M}_D\mathbf{y}$ and $\widetilde{\mathbf{X}} \equiv \mathbf{M}_D\mathbf{X}$, and $\widetilde{\mathbf{X}}'\mathbf{W}\widetilde{\mathbf{X}}$ is nonsingular for the retained columns, then the slope coefficients are obtained from the low-dimensional regression

$$\widehat{\boldsymbol{\beta}} = (\widetilde{\mathbf{X}}'\mathbf{W}\widetilde{\mathbf{X}})^{-1}\widetilde{\mathbf{X}}'\mathbf{W}\widetilde{\mathbf{y}} \tag{2}$$

The final regression has only K slope variables. The expensive numerical step is applying $\mathbf{M}_D$ to the outcome and to every regressor. `xhdfe` therefore improves runtime by accelerating this residualization step while preserving the same coefficient target.

3.2 Multiple fixed effects

For one categorical fixed effect, residualization is weighted group demeaning: compute the group mean of each column of $\mathbf{Z} \equiv [\mathbf{y}, \mathbf{X}]$, then subtract that mean from every observation in the group. With several fixed-effect dimensions, one pass of group demeaning is not enough because the fixed-effect subspaces are not generally orthogonal. Demeaning by worker fixed effects, for example, can reintroduce firm-level means, and demeaning by firm fixed effects can reintroduce worker-level means. The standard computational approach is to iterate one-way residual makers until the transformed variables stop changing.

Let $\mathbf{M}_g$ denote the residual maker for fixed-effect dimension g, applied to all columns of $\mathbf{Z} \equiv [\mathbf{y}, \mathbf{X}]$. A Gauss-Seidel alternating-projection sweep, sometimes called a Kaczmarz transform in this context, is

$$T_{\text{GS}}(\mathbf{Z}) = \mathbf{M}_G\mathbf{M}_{G-1}\cdots\mathbf{M}_1\mathbf{Z}$$

A symmetric sweep, sometimes denoted the symmetric Kaczmarz transform, applies a forward and backward pass:

$$T_{\text{sym}}(\mathbf{Z}) = \mathbf{M}_1\mathbf{M}_2\cdots\mathbf{M}_{G-1}\mathbf{M}_G\mathbf{M}_{G-1}\cdots\mathbf{M}_2\mathbf{M}_1\mathbf{Z}$$

Residualizing with multiple fixed effects can therefore be written as a fixed-point problem. For a chosen sweep operator T, the absorbed matrix is the limit reached by iterating from $\mathbf{Z}$. It satisfies

$$\mathbf{Z}^* = T(\mathbf{Z}^*) \qquad \mathbf{Z}^* = \mathbf{M}_D\mathbf{Z} \tag{3}$$

The role of the absorber is to solve this fixed-point problem numerically. It starts from the untransformed matrix and applies repeated sweeps,

$$\mathbf{Z}^{(m+1)} = T(\mathbf{Z}^{(m)}) \qquad \mathbf{Z}^{(0)} = \mathbf{Z}$$

until the change in the transformed columns is below the requested tolerance. With strictly positive weights and the $\mathbf{M}_g$ defined as $\mathbf{W}$-orthogonal residual makers, finite-dimensional alternating-projection theory implies that this sequence converges from $\mathbf{Z}$ to $\mathbf{M}_D\mathbf{Z}$.

This fixed-point formulation matters for interpretation. Both `reghdfe` and `xhdfe` target the same partialled-out regression in (2). `reghdfe`'s default high-dimensional fixed-effect routine is already algorithmically sophisticated, using the method of alternating projections with a symmetric Kaczmarz transform and conjugate-gradient acceleration. The main distinction is that `xhdfe` moves the projection work into a compiled C++ backend and adds execution choices that reduce either the number of sweeps or the cost of each sweep.

3.3 High-level algorithm

The core computation has three setup stages before absorption. First, `xhdfe` constructs the estimation sample and applies the requested singleton rule. Second, each absorbed fixed-effect variable is normalized into compact integer group identifiers. This converts arbitrary Stata variables into group-index arrays that the backend can use repeatedly. Third, the program prepares group metadata: group counts, group pointers or group-ID arrays, weight sums, reciprocals of weight sums, and simple profile statistics about the fixed-effect dimensions.

The absorber then chooses an absorption method and an order in which to visit the fixed-effect dimensions. It iterates over $\mathbf{Z} \equiv [\mathbf{y}, \mathbf{X}]$, computing group means and subtracting them from all transformed columns in place. Depending on the selected method, a sweep may be unidirectional, symmetric, or based on a simultaneous update. The backend may also use extrapolation steps to move the fixed-point sequence closer to its limit. Once convergence is reached, `xhdfe` estimates the low-dimensional regression in (2), computes the variance-covariance matrix and degrees of freedom, and posts Stata e-class results.

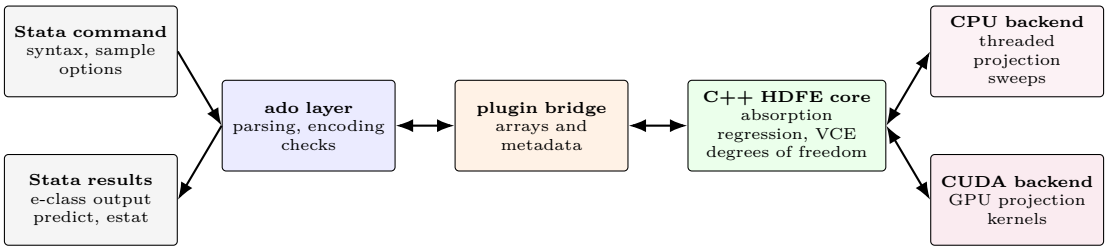
This decomposition separates the econometric problem from the systems problem. The econometric object is the partialled-out regression. The systems problem is how cheaply the software can compute the transformed matrix $(\tilde{\mathbf{y}}, \widetilde{\mathbf{X}})$. On large datasets, that cost is driven by memory traffic, temporary allocation, sweep count, thread scheduling, and the ability to keep fixed-effect operations close to the data layout.

3.4 Software architecture

The software architecture follows directly from this decomposition. The Stata layer should preserve the interface that applied researchers use, while the compiled layer should do the repeated projection work. Figure 2 summarizes the resulting division of labor.

`xhdfe` has three layers. The ado layer implements the Stata interface by parsing syntax, marking the estimation sample, encoding string fixed effects, materializing categorical interactions, checking unsupported syntax, and posting e-class results. The plugin layer transfers Stata arrays and configuration data to compiled code. The C++ layer performs fixed-effect absorption, singleton dropping, regression, covariance estimation, degrees-of-freedom accounting,

Figure 2: `xhdfe` keeps the Stata interface while moving absorption to compiled backends



Notes: The ado layer parses the command, prepares the estimation sample, and posts the results in Stata. The plugin transfers variables, fixed-effect identifiers, weights, and options to compiled code and returns the computed results. CPU and CUDA are alternative execution choices for the same absorption problem. They differ in how the projection sweeps are executed.

fixed-effect recovery, and backend metadata.

This architecture preserves the main advantages of Stata software: users remain inside Stata, results are posted in e-class form, and postestimation commands can follow Stata conventions. At the same time, the expensive numerical work is not constrained to ado or Mata code. The absorber can use CPU threading. When built with CUDA, selected work can execute on a GPU. This design improves runtime in two ways. The numerical method can reduce the number of projection sweeps, while threading and CUDA reduce the cost of each sweep.

3.5 Runtime mechanisms

Runtime depends on two margins: the number of projection sweeps needed for convergence and the cost of each sweep. In both cases, the target remains the same fixed point $\mathbf{M}_D\mathbf{Z}$ from (3). The tables shown below are single-run decompositions intended to isolate related mechanisms that improve overall runtime.

Sweep order and method choice. Conditional on a sweep-based method, the program must choose the order in which the fixed-effect dimensions are visited. For a candidate order π, a forward sweep applies

$$T_\pi(\mathbf{Z}) = \mathbf{M}_{\pi(G)} \cdots \mathbf{M}_{\pi(2)}\mathbf{M}_{\pi(1)}\mathbf{Z}$$

This sequencing does not change the limiting projection $\mathbf{M}_D\mathbf{Z}$, but it can change how quickly the fixed-point iteration contracts. A poor order can therefore require more full passes over the data. For designs with more than two fixed-effect dimensions, `xhdfe` uses low-cost profile information, such as the number of groups, average group size, group-size dispersion, and singleton prevalence, to choose an order. With two dimensions, it compares the two candidate orders using two trial sweeps on at most the first 200,000 observations of the outcome and

selects

$$\min_{\pi} \left\| T_{\pi}^{2}(\mathbf{y}_{\text{probe}}) \right\|$$

A separate choice is the numerical update used to compute $\mathbf{M}_D\mathbf{Z}$ in the first place. Gauss-Seidel uses the newest residualized values immediately,

$$\mathbf{Z} \leftarrow \mathbf{M}_1\mathbf{Z} \quad \mathbf{Z} \leftarrow \mathbf{M}_2\mathbf{Z} \quad \dots \quad \mathbf{Z} \leftarrow \mathbf{M}_G\mathbf{Z}$$

Symmetric Gauss-Seidel applies a forward and backward pass, which can improve stability but increases the amount of work per logical iteration. Jacobi-style updates compute corrections from a common current state and apply a simultaneous relaxed update, using the one-way projections $\mathbf{P}_g = \mathbf{I} - \mathbf{M}_g$ onto the group means of dimension g,

$$\mathbf{Z}^{(m+1)} = \mathbf{Z}^{(m)} - \omega \sum_{g=1}^{G} \mathbf{P}_g \mathbf{Z}^{(m)}$$

This structure is easier to parallelize across fixed-effect dimensions, but it can require many more iterations. Table 1 illustrates this margin on the restricted QP benchmark specification introduced in Section 2.

Table 1: Absorption method and backend in the QP benchmark

Backend	Method	Seconds	Iterations
CPU	Auto	43.053	42
CPU	Gauss-Seidel	45.264	42
CPU	Jacobi	357.308	916
CUDA	Auto	13.535	42
CUDA	Gauss-Seidel	16.018	42
CUDA	Jacobi	23.013	916

Notes: Restricted QP benchmark specification with `tolerance(1e-8)` and `tolerancemode(xhdfe-fast)`. CUDA entries request `gpubackend(cuda)`. Iterations are multiway fixed-effects (MWFE) convergence iterations. Each entry is measured once.

The automatic setting chooses an absorption method without changing the within-transformed variables. In this benchmark, `absorptionmethod(auto)` selected Gauss-Seidel. Running Jacobi on the CUDA backend instead of the CPU reduced wall time from 357.308 seconds to 23.013 seconds, but did not change the iteration count: it still required 916 iterations. CUDA automatic absorption converged in 42 iterations and took 13.535 seconds. These results separate two mechanisms: hardware parallelism reduces the cost of a pass, while the absorption method determines how many effective passes are required.

Extrapolated sweeps. Alternating projections form a fixed-point iteration. If T is the selected sweep operator and $\mathbf{Z}^*$ is the absorbed matrix, then $\mathbf{Z}^* = T(\mathbf{Z}^*)$. Plain iteration repeatedly applies T, but convergence can be accelerated by using recent iterates to estimate where the sequence is heading. `xhdfe`'s C++ core includes an Irons-Tuck/Aitken-style extrapolation step. For a current state $\mathbf{Z}$, let $T\mathbf{Z} \equiv T(\mathbf{Z})$ and $T^2\mathbf{Z} \equiv T(T\mathbf{Z})$ denote one and two applications of the sweep operator, and define

$$\boldsymbol{\Delta} = T^2\mathbf{Z} - T\mathbf{Z} \qquad \boldsymbol{\Delta}^2 = T^2\mathbf{Z} - 2T\mathbf{Z} + \mathbf{Z}$$

The backend computes a scalar

$$\gamma = \frac{\left\langle \boldsymbol{\Delta}, \boldsymbol{\Delta}^2 \right\rangle}{\left\| \boldsymbol{\Delta}^2 \right\|^2}$$

where, for conformable matrices $\mathbf{A}$ and $\mathbf{B}$, the inner product is taken over all observations i and transformed columns j:

$$\langle \mathbf{A}, \mathbf{B} \rangle = \sum_j \sum_i A_{ij} B_{ij} \qquad \|\mathbf{A}\|^2 = \langle \mathbf{A}, \mathbf{A} \rangle$$

The numerator $\left\langle \boldsymbol{\Delta}, \boldsymbol{\Delta}^2 \right\rangle$ is the projection cross-product, and the denominator $\left\| \boldsymbol{\Delta}^2 \right\|^2$ is the squared length of the direction being projected onto. Thus γ is the least-squares scalar coefficient from projecting $\boldsymbol{\Delta}$ onto $\boldsymbol{\Delta}^2$. The backend then updates

$$\mathbf{Z}_{\text{acc}} = T^2\mathbf{Z} - \gamma(T^2\mathbf{Z} - T\mathbf{Z})$$

This update does not change the target projection. It attempts to reach the target in fewer demeaning sweeps. The QP benchmark isolates this mechanism by comparing the standard extrapolated absorber with an otherwise identical run in which extrapolation is disabled. Table 2 reports the same restricted QP benchmark specification.

Table 2: Effect of disabling extrapolated sweeps

Backend	Method	Baseline (s)	No extrapolation (s)	Iterations
CPU	Gauss-Seidel	42.720	76.447	$42 \to 246$
CPU	Symmetric	59.519	169.059	$42 \to 312$
CUDA	Gauss-Seidel	13.691	18.485	$42 \to 246$
CUDA	Symmetric	14.764	19.888	$42 \to 312$

Notes: The no-extrapolation condition turns off extrapolation while leaving the absorption method and backend unchanged. All rows use `tolerance(1e-8)` and `tolerancemode(xhdfe-fast)`. Each setting is measured once.

The results show that extrapolation mainly reduces the number of effective sweeps, while the backend determines the cost of each extra sweep. Disabling extrapolation raises the iteration count by the same amount on CPU and CUDA, but the timing penalty is smaller on CUDA because each additional projection pass is cheaper. For Gauss-Seidel, disabling extrapolation adds about 34 seconds on CPU and 5 seconds on CUDA; for symmetric sweeps, the corresponding penalties are about 110 and 5 seconds.

Matrix-free MLSMR absorption. Projection sweeps are not the only way to compute the within transformation. For standard fixed effects, absorption can also be written as the auxiliary least-squares problem of finding the fixed-effect contribution that best explains each column of $\mathbf{Z} \equiv [\mathbf{y}, \mathbf{X}]$. Let $\mathbf{D} \in \mathbb{R}^{n \times q}$ denote the standard fixed-effect dummy design, and let $\mathbf{A} \in \mathbb{R}^{q \times (K+1)}$ collect the fixed-effect coefficients associated with the columns of $\mathbf{Z}$. Then

$$\widehat{\mathbf{A}}_{\mathbf{Z}} \in \arg\min_{\mathbf{A}} \operatorname{tr}\{(\mathbf{Z} - \mathbf{D}\mathbf{A})'\mathbf{W}(\mathbf{Z} - \mathbf{D}\mathbf{A})\} \qquad \widehat{\mathbf{A}}_{\mathbf{Z}} = (\mathbf{D}'\mathbf{W}\mathbf{D})^{+}\mathbf{D}'\mathbf{W}\mathbf{Z}$$

for a generalized inverse $(\cdot)^+$. By the definition of $\mathbf{P}_D$ and $\mathbf{M}_D$ in (1),

$$\mathbf{D}\widehat{\mathbf{A}}_{\mathbf{Z}} = \mathbf{P}_D\mathbf{Z} \qquad \mathbf{M}_D\mathbf{Z} = \mathbf{Z} - \mathbf{D}\widehat{\mathbf{A}}_{\mathbf{Z}}$$

Because $\mathbf{Z} \equiv [\mathbf{y}, \mathbf{X}]$, the absorbed matrix is

$$\mathbf{M}_D\mathbf{Z} = [\mathbf{M}_D\mathbf{y}, \mathbf{M}_D\mathbf{X}] = [\widetilde{\mathbf{y}}, \widetilde{\mathbf{X}}]$$

Once this matrix has been computed, $\widehat{\boldsymbol{\beta}}$ is recovered from the FWL regression in (2).

This formulation suggests Krylov methods such as LSMR.[5] Krylov methods solve large linear least-squares problems through repeated products with the design matrix and its transpose. They avoid forming the full matrix or normal equations and do not require a direct factorization. In this setting, the dummy matrix $\mathbf{D}$ is still never formed: products with $\mathbf{D}$ assign group coefficients to observations, while products with $\mathbf{D}'$ compute weighted group sums. LSMR is a Krylov least-squares algorithm that uses these two operations to update the fixed-effect coefficient vector. It is therefore a natural candidate for computing the same projection $\mathbf{M}_D\mathbf{Z}$ when repeated demeaning sweeps contract slowly.

`xhdfe` includes a modified LSMR absorber for standard fixed effects. The implementation is matrix-free and uses preconditioning based on the fixed-effect structure, including local factor-pair solves when they are cheap enough. A closely related graph-preconditioned Krylov solver for high-dimensional fixed-effects demeaning, whose reusable preconditioner is likewise built from local factor-pair subproblems, is developed by Fischer and Schröder (2026). The modified LSMR path changes the numerical route to $\mathbf{M}_D\mathbf{Z}$, not the econometric target. It can be useful when ordinary alternating projections contract slowly, because the Krylov iteration uses information from previous search directions rather than relying only on repeated demeaning sweeps.

Compiled CPU kernels and GPU absorption. Each demeaning pass is simple mathematically but demanding computationally. For each fixed-effect dimension and each transformed column, the program must read group identifiers, accumulate group sums, divide by group weights, and subtract the corresponding group mean from every observation. `xhdfe` implements these operations in compiled routines that update the working residual matrix directly. The C++ backend uses compact integer group identifiers, precomputed weight reciprocals, thread-local accumulation buffers, sparse-reset logic, and specialized kernels for common small-K cases. Where the group structure is favorable, compressed group pointers allow the program to scan observations group by group and parallelize across groups.

CPU threading reduces the cost per sweep by parallelizing group reductions and subtraction passes. The GPU backend targets the same operation. It moves the repeated group-sum and mean-subtraction kernels to CUDA when a CUDA-enabled device is available. GPU execution is most useful when the data, fixed-effect structure, or number of repeated sweeps is large enough to offset data-preparation and kernel-launch overhead. CUDA lowers the

[5]For the LSMR algorithm, see Fong and Saunders (2011). Paige and Saunders (1982) provide the closely related LSQR least-squares algorithm, and Saad (2003) gives a general treatment of Krylov methods and sparse iterative solvers.

cost of projection passes. It does not by itself improve the contraction rate of the chosen absorption map.

Table 3 shows this cost-per-sweep margin on the QP specification. The iteration count is unchanged across the three configurations, so the timing gain comes from reducing the cost of each projection pass rather than from changing the fixed-point problem.

Table 3: Computational mechanisms on the QP specification

Setting	Seconds	Iterations	Mechanism
`xhdfe` CPU, 1 thread	192.427	42	Single-thread compiled absorption
`xhdfe` CPU, 16 threads	39.840	42	OpenMP parallel absorption
`xhdfe` CUDA auto	13.992	42	GPU absorption backend

Notes: Each displayed setting is measured once on the restricted QP specification with `tolerance(1e-8)` and `tolerancemode(xhdfe-fast)`. Iterations are multiway fixed-effects (MWFE) convergence iterations.

Relative to the one-thread CPU run, 16-thread absorption reduces wall time from 192.427 to 39.840 seconds, a 79.3% reduction. CUDA reduces it further to 13.992 seconds, 92.7% below the one-thread timing, without changing the iteration count.

3.6 Interpretation

The architecture analyzed here changes the speed of the within transformation, not the econometric estimand. The estimator remains the FWL regression in (2). Runtime improves because $\mathbf{M}_D\mathbf{Z}$ is computed through projection orders and update methods that converge in fewer sweeps, extrapolation of the fixed-point iteration, compiled and parallel group operations, and CUDA execution when the projection passes are large enough. Single-thread compiled execution can therefore already be much faster than `reghdfe`, while CPU threading and GPU execution reduce the cost further.

Section 4 presents the Stata syntax and options for ordinary and heterogeneous-slope models. It also describes the numerical and backend controls and the available postestimation output. Section 5 then uses two large wage regressions to assess numerical agreement with `reghdfe`, measure the runtime gains, and compare `xhdfe` with alternative HDFE implementations.

4 Syntax and use of `xhdfe`

The Stata command `xhdfe` follows the syntax conventions of `reghdfe` for the common linear fixed-effects case. The basic syntax is:

```
xhdfe depvar [indepvars] [if] [in] [weight],
    absorb(absvars [, savefe]) [savefes(prefix)]
    [vce(vcetype) robust cluster(varlist) residuals(newvar)]
    [keepsingletons noconstant tolerance(#) tolerancemode(str)]
```

[**convergence**(*str*)]
[**fetolerance**(#) **ferecoverymethod**(*str*) **maxiter**(#) **iterate**(#)]
[**absorptionmethod**(*str*) **symmetricsweep** **jacobirelaxation**(#)]
[**gpubackend**(*str*) **numthreads**(#) **dofadjustments**(*list*)]

where *depvar* is the dependent variable, *indepvars* is the list of regressors, and *absvars* gives the fixed effects to be absorbed. Absorbed variables may include heterogeneous slopes: `fe#c.x` absorbs slopes in continuous `x`, while `fe##c.x` also absorbs the group fixed effect.

Optional material appears in square brackets. Underlined letters mark the minimum abbreviation accepted. For example, `absorb()` may be typed as `a()`, while `numthreads()` may be typed as `numt()`. The shortest forms `a()`, `g()`, `i()`, and `iter()` are exact aliases, so intermediate forms such as `ab()` or `gr()` are not accepted. The displayed syntax is not exhaustive. `xhdfe` also includes display and small-sample-adjustment options documented in the help file. Weights use standard Stata syntax. For the `reghdfe`-compatible interface documented in this article, `xhdfe` supports `aweight`, `fweight`, `pweight`, and `iweight`. With `pweight` and no explicit `vce()`, `xhdfe` uses robust standard errors.

A related syntax covers group-level outcomes, with or without individual fixed effects.

xhdfe *depvar* [*indepvars*] [*if*] [*in*] [*weight*],
group(*groupvar*) [**individual**(*indvar*)] [**absorb**(*absvars*)]
[**aggregation**(*str*) *options*]

The same estimation, inference, and backend options apply unless explicitly excluded. Heterogeneous slopes are unavailable in `group()` mode, while `savefe`/`savefes()` are unavailable when both `group()` and `individual()` are specified. The `individual()` variable is appended to `absorb()` if needed. The option `i()` is accepted as shorthand. The default `aggregation(mean)` can also be written `aggregation(avg)` or `aggregation(average)`. The validation and performance evidence in Section 5 focuses on observation-level HDFE regressions. The group-outcome syntax is documented here for completeness.

4.1 Minimal example

The examples use the public `nlswork` data distributed with Stata. A minimal worker-year specification absorbs worker and year fixed effects:

```
. webuse nlswork, clear
(National Longitudinal Survey of Young Women, 14-24 years old in 1968)
. xhdfe ln_wage ttl_exp tenure union, absorb(idcode year)
(dropped 673 singleton observations)
(MWFE estimator converged in 6 iterations)
HDFE Linear regression                          Number of obs    =     18,337
Absorbing 2 HDFE group(s)                       F(   3,   14862) =     360.52
Statistics based on homoskedasticity            Prob > F         =     0.0000
                                                R-squared        =     0.7587
                                                Adj R-squared    =     0.7023
                                                Within R-sq.     =     0.0678
```

```
                                                Root MSE        =     0.2531
```

ln_wage	Coefficient	Std. err.	t	P>|t|	[95% conf.	interval]
ttl_exp	.0358447	.0018439	19.44	0.000	.0322304	.0394589
tenure	.0085512	.0009207	9.29	0.000	.0067464	.0103559
union	.09983	.0069307	14.40	0.000	.086245	.1134149
_cons	1.422448	.0132771	107.14	0.000	1.396423	1.448472

Absorbed degrees of freedom:

Absorbed FE	Categories	- Redundant	= Num. Coefs
idcode	3461	0	3461
year	12	1	11

Additional fixed effects are added by listing them in `absorb()`:

```
. xhdfe ln_wage ttl_exp union, absorb(idcode ind_code occ_code year)
(output omitted)
```

Heterogeneous slopes can also be absorbed with continuous factor-variable notation inside `absorb()`. For example, `occ_code##c.ttl_exp` absorbs occupation fixed effects and allows the coefficient on experience to vary by occupation:

```
. xhdfe ln_wage tenure union, absorb(idcode year occ_code##c.ttl_exp)
(dropped 672 singleton observations)
(MWFE estimator converged in 248 iterations)
HDFE Linear regression                            Number of obs    =     18,265
Absorbing 3 HDFE group(s)                         F(   2,   14773) =     144.87
Statistics based on homoskedasticity              Prob > F         =     0.0000
                                                  R-squared        =     0.7681
                                                  Adj R-squared    =     0.7133
                                                  Within R-sq.     =     0.0192
                                                  Root MSE         =     0.2483
```

ln_wage	Coefficient	Std. err.	t	P>|t|	[95% conf.	interval]
tenure	.0089118	.0009114	9.78	0.000	.0071253	.0106983
union	.0932063	.0068705	13.57	0.000	.0797393	.1066733
_cons	1.704113	.0043803	389.04	0.000	1.695527	1.712699

Absorbed degrees of freedom:

Absorbed FE	Categories	- Redundant	= Num. Coefs	
idcode	3454	0	3454	
year	12	1	11	
occ_code	13	1	12	?
occ_code#c.ttl_exp	13	0	13	?

? = number of redundant parameters may be higher

4.2 Weights

`xhdfe` accepts the standard Stata weight types (`aweight`, `fweight`, `pweight`, and `iweight`) with the syntax and reporting conventions of `reghdfe`. The following example uses analytic weights equal to usual hours worked:

```
. webuse nlswork, clear
(National Longitudinal Survey of Young Women, 14-24 years old in 1968)
. xhdfe ln_wage ttl_exp tenure union [aweight=hours], absorb(idcode year)
(dropped 675 singleton observations)
(MWFE estimator converged in 19 iterations)
HDFE Linear regression                            Number of obs    =     18,301
Absorbing 2 HDFE group(s)                         F(   3,   14828) =     378.27
Statistics based on homoskedasticity              Prob > F         =     0.0000
                                                  R-squared        =     0.7753
                                                  Adj R-squared    =     0.7227
                                                  Within R-sq.     =     0.0711
                                                  Root MSE         =     0.2379

------------------------------------------------------------------------------
     ln_wage | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
     ttl_exp |   .0382148   .0018136    21.07   0.000     .0346599    .0417698
      tenure |   .0073468   .0008606     8.54   0.000     .0056599    .0090336
       union |   .0953244   .0065539    14.54   0.000     .0824779    .1081709
       _cons |   1.413652   .0133838   105.62   0.000     1.387418    1.439886
------------------------------------------------------------------------------

Absorbed degrees of freedom:
-----------------------------------------------------+
 Absorbed FE | Categories  - Redundant  = Num. Coefs |
-------------+---------------------------------------|
      idcode |       3459           0        3459   |
        year |         12           1          11   |
-----------------------------------------------------+
```

With `pweight` and no explicit `vce()`, `xhdfe` reports robust standard errors, matching `reghdfe`. Before estimation, the command validates analytic weights: negative or nonfinite values are rejected, while zero-weight observations remain allowed and contribute nothing to weighted moments.

4.3 Robust and clustered standard errors

The variance estimator is controlled with `vce()`. The command supports unadjusted, robust, and clustered standard errors:

```
. xhdfe ln_wage ttl_exp tenure union, absorb(idcode year) vce(robust)
(dropped 673 singleton observations)
(MWFE estimator converged in 6 iterations)
HDFE Linear regression                            Number of obs    =     18,337
Absorbing 2 HDFE group(s)                         F(   3,   14862) =     299.87
Statistics robust to heteroskedasticity           Prob > F         =     0.0000
                                                  R-squared        =     0.7587
                                                  Adj R-squared    =     0.7023
                                                  Within R-sq.     =     0.0678
                                                  Root MSE         =     0.2531

------------------------------------------------------------------------------
             |               Robust
     ln_wage | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
     ttl_exp |   .0358447   .0021547    16.64   0.000     .0316212    .0400681
      tenure |   .0085512   .0010343     8.27   0.000     .0065238    .0105785
       union |     .09983   .0079942    12.49   0.000     .0841603    .1154996
       _cons |   1.422448   .0151909    93.64   0.000     1.392671    1.452224
------------------------------------------------------------------------------

Absorbed degrees of freedom:
-----------------------------------------------------+
 Absorbed FE | Categories  - Redundant  = Num. Coefs |
-------------+---------------------------------------|
```

```
      idcode |       3461          0       3461
        year |         12          1         11
```

Clustered inference can be requested with either `vce(cluster ...)` or the `cluster()` shorthand:

```
. xhdfe ln_wage ttl_exp tenure union, absorb(idcode year) vce(cluster idcode)
(dropped 673 singleton observations)
(MWFE estimator converged in 6 iterations)
HDFE Linear regression                            Number of obs    =     18,337
Absorbing 2 HDFE group(s)                         F(   3,    3460) =     169.34
Statistics robust to heteroskedasticity           Prob > F         =     0.0000
                                                  R-squared        =     0.7587
                                                  Adj R-squared    =     0.7023
                                                  Within R-sq.     =     0.0678
                                                  Root MSE         =     0.2531
                                (Std. err. adjusted for 3,461 clusters in idcode)
------------------------------------------------------------------------------
             |               Robust
     ln_wage | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
     ttl_exp |   .0358447    .002835    12.64   0.000     .0302862    .0414031
      tenure |   .0085512   .0013316     6.42   0.000     .0059404    .0111619
       union |     .09983   .0094596    10.55   0.000      .081283    .1183769
       _cons |   1.422448   .0200455    70.96   0.000     1.383145     1.46175
------------------------------------------------------------------------------

Absorbed degrees of freedom:
-----------------------------------------------------------------------------
 Absorbed FE | Categories  - Redundant  = Num. Coefs                          |
-------------+---------------------------------------------------------------|
      idcode |       3461        3461           0    *                      |
        year |         12           1          11                           |
-----------------------------------------------------------------------------
* = FE nested within cluster; treated as redundant for DoF computation
```

The option `cluster(idcode)` is equivalent to `vce(cluster idcode)`. Cluster variables may include interactions specified with `#` or `##`. Multiway clustering is accepted by listing more than one cluster variable inside `vce(cluster ...)`, for example:

```
. xhdfe ln_wage ttl_exp, absorb(idcode year) vce(cluster idcode ind_code#year)
(dropped 544 singleton observations)
(MWFE estimator converged in 6 iterations)
HDFE Linear regression                            Number of obs    =     27,649
Absorbing 2 HDFE group(s)                         F(   1,     178) =     378.34
Statistics robust to heteroskedasticity           Prob > F         =     0.0000
                                                  R-squared        =     0.6707
                                                  Adj R-squared    =     0.6123
                                                  Within R-sq.     =     0.0424
                                                  Root MSE         =     0.2966
                          (Std. err. adjusted for 179 clusters in idcode ind_code#year)
------------------------------------------------------------------------------
             |               Robust
     ln_wage | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
     ttl_exp |   .0418225   .0021502    19.45   0.000     .0375795    .0460656
       _cons |   1.416159   .0134425   105.35   0.000     1.389632    1.442687
------------------------------------------------------------------------------

Absorbed degrees of freedom:
-----------------------------------------------------------------------------
 Absorbed FE | Categories  - Redundant  = Num. Coefs                          |
-------------+---------------------------------------------------------------|
      idcode |       4151        4151           0    *                      |
        year |         15           1          14                           |
-----------------------------------------------------------------------------
```

```
* = FE nested within cluster; treated as redundant for DoF computation
```

For multiway clustering, the headline cluster count follows the `reghdfe` convention and reports the limiting cluster count used for inference. Dimension-specific cluster counts are stored separately.

4.4 Backend options

The options `numthreads()` and `gpubackend()` control how the absorber is executed. To request the CPU backend with `numthreads(8)`, type

```
. webuse nlswork, clear
(National Longitudinal Survey of Young Women, 14-24 years old in 1968)
. xhdfe ln_wage ttl_exp, absorb(idcode year) numthreads(8) gpubackend(cpu)
(dropped 547 singleton observations)
(MWFE estimator converged in 6 iterations)
HDFE Linear regression                            Number of obs    =     27,987
Absorbing 2 HDFE group(s)                         F(   1,   23808) =   1053.99
Statistics based on homoskedasticity              Prob > F         =    0.0000
                                                  R-squared        =    0.6701
                                                  Adj R-squared    =    0.6122
                                                  Within R-sq.     =    0.0424
                                                  Root MSE         =    0.2965

------------------------------------------------------------------------------
     ln_wage | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
     ttl_exp |   .0416716   .0012836    32.47   0.000     .0391557    .0441875
       _cons |   1.416666   .0082239   172.26   0.000     1.400547    1.432786
------------------------------------------------------------------------------

Absorbed degrees of freedom:
-----------------------------------------------------------
 Absorbed FE | Categories  - Redundant  = Num. Coefs
-------------+---------------------------------------------
      idcode |       4164           0        4164
        year |         15           1          14
-----------------------------------------------------------

. display "threads used = " e(threads_used)
threads used = 8
. display "GPU used = " e(gpu_used)
GPU used = 0
. display "backend = `e(gpu_backend)´"
backend = cpu
```

When a CUDA-enabled plugin and CUDA device are available, the following example requests CUDA and prints backend diagnostics:

```
. webuse nlswork, clear
(National Longitudinal Survey of Young Women, 14-24 years old in 1968)
. xhdfe ln_wage ttl_exp, absorb(idcode year) numthreads(8) gpubackend(cuda)
(dropped 547 singleton observations)
(MWFE estimator converged in 6 iterations)
HDFE Linear regression                            Number of obs    =     27,987
Absorbing 2 HDFE group(s)                         F(   1,   23808) =   1053.99
Statistics based on homoskedasticity              Prob > F         =    0.0000
                                                  R-squared        =    0.6701
                                                  Adj R-squared    =    0.6122
                                                  Within R-sq.     =    0.0424
                                                  Root MSE         =    0.2965

-------------+----------------------------------------------------------------
```

ln_wage	Coefficient	Std. err.	t	P>|t|	[95% conf.	interval]
ttl_exp	.0416716	.0012836	32.47	0.000	.0391557	.0441875
_cons	1.416666	.0082239	172.26	0.000	1.400547	1.432786

Absorbed degrees of freedom:

Absorbed FE	Categories	- Redundant	= Num. Coefs
idcode	4164	0	4164
year	15	1	14

```
. display "threads used = " e(threads_used)
threads used = 8
. display "GPU used = " e(gpu_used)
GPU used = 1
. display "backend = `e(gpu_backend)´"
backend = cuda
```

After estimation, `e(threads_used)` reports the number of threads, `e(gpu_used)` indicates whether the GPU was used, and `e(gpu_backend)` identifies the backend. In the examples above, both backends report that all eight requested threads were used, while only the CUDA example reports GPU execution. If a non-CPU backend is requested but is not available in the active plugin build, `xhdfe` stops with an error rather than silently returning CPU output. The prebuilt plugins distributed with the package are CPU-only. On a Linux machine with an NVIDIA GPU, the companion command `xhdfegpu` builds a CUDA-enabled plugin matched to the local GPU and installs it in place of the CPU plugin in one step, after which CUDA execution is requested with `gpubackend(cuda)`. The `xhdfegpu` help file documents the requirements.

4.5 Absorption options

The option `absorptionmethod()` selects the iterative method used for the fixed-effect absorption step. The default `absorptionmethod(auto)` is the recommended setting for ordinary use. On eligible standard-FE designs run on the CPU, it first applies a low-cost diagnostic and may select modified LSMR when ordinary projection sweeps are expected to converge slowly. Otherwise, it uses a sweep-based method. Manual choices such as `gauss-seidel`, `symmetric` or `symmetric-gauss-seidel`, and `jacobi` are mainly useful for benchmarking, robustness checks, or reproducing computational experiments.

The explicit `absorptionmethod(auto-mlsmr)` option requests the modified-LSMR selector directly. The opt-in `lsmr` and `mlsmr` methods are CPU-only and support standard fixed effects; `mlsmr` is also available as `within`. These methods do not support heterogeneous slopes, `group()`/`individual()` models, or fixed-effect recovery with `savefe`/`savefes()`. The related options `symmetricsweep` and `jacobirelaxation(#)` further control the symmetric sweep and Jacobi relaxation parameter.

If the default is overridden, the method should be reported with the backend, tolerance mode, and thread settings. The stored diagnostic `e(absorption_method_used)` is numeric. For the methods discussed here, 1 is `gauss-seidel`, 2 is `symmetric-gauss-seidel`, 3 is `jacobi`, 5 is `lsmr`, and 6 is `mlsmr`. The help file documents the full code list, including 4 for

the Schwarz/conjugate-gradient path.

```
. webuse nlswork, clear
(National Longitudinal Survey of Young Women, 14-24 years old in 1968)
. xhdfe ln_wage ttl_exp, absorb(idcode year) absorptionmethod(symmetric)
(dropped 547 singleton observations)
(MWFE estimator converged in 6 iterations)
HDFE Linear regression                            Number of obs    =     27,987
Absorbing 2 HDFE group(s)                         F(   1,   23808) =   1053.99
Statistics based on homoskedasticity              Prob > F         =    0.0000
                                                  R-squared        =    0.6701
                                                  Adj R-squared    =    0.6122
                                                  Within R-sq.     =    0.0424
                                                  Root MSE         =    0.2965
------------------------------------------------------------------------------
     ln_wage | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
     ttl_exp |   .0416716   .0012836    32.47   0.000     .0391557    .0441875
       _cons |   1.416666   .0082239   172.26   0.000     1.400547    1.432786
------------------------------------------------------------------------------
Absorbed degrees of freedom:
-----------------------------------------------------+
 Absorbed FE | Categories  - Redundant  = Num. Coefs |
-------------+---------------------------------------|
      idcode |       4164           0        4164   |
        year |         15           1          14   |
-----------------------------------------------------+
. display "absorption method used = " e(absorption_method_used)
absorption method used = 2
```

4.6 Saving fixed effects, residuals, and fitted values

`xhdfe` can save residuals at estimation time and can save the contributions of the absorbed fixed effects. The prefix-based form is

```
. xhdfe ln_wage ttl_exp, absorb(idcode year) residuals(u) savefes(fe_)
(output omitted)
```

The command stores regression residuals in `u` and fixed-effect contributions in variables named from the `fe_` prefix. The alternative `absorb(..., savefe)` saves fixed-effect contributions using the `__hdfe1__`, `__hdfe2__`, ... naming convention:

```
. xhdfe ln_wage ttl_exp, absorb(idcode year, savefe) residuals(u)
(output omitted)
```

For heterogeneous slopes, `savefe` and `savefes()` follow the same logic but add slope-coefficient variables. A term `g##c.z` absorbs both a group intercept and a group-specific slope in `z`. With `savefes(fe_)`, `xhdfe` saves one variable for the base group effect and another with the suffix `Slope1` for the group-specific slope coefficient. A term `g#c.z` absorbs only the slope, so only the `Slope1` variable is saved. For example,

```
. xhdfe ln_wage tenure, absorb(occ_code##c.ttl_exp) residuals(u) savefes(fe_)
(output omitted)
. describe fe_occ_code__c_ttl_exp*

Variable      Storage   Display    Value
    name         type    format    label      Variable label
------------------------------------------------------------------------------
fe_occ_code__~p double  %10.0g                [FE] 1.occ_code##c.ttl_exp
fe_occ_code__~1 double  %10.0g                [FE] 1.occ_code#c.ttl_exp
```

saves `fe_occ_code__c_ttl_exp` for the occupation fixed effect. The occupation-specific coefficient on `ttl_exp` is saved in the corresponding `Slope1` variable, formed by appending `Slope1` to the base saved-effect name. The observation-level contribution of this heterogeneous-slope term is the saved fixed effect plus the saved slope coefficient multiplied by `ttl_exp`. Because fixed effects are identified only up to normalization, the level of an individual saved component can depend on the normalization used. The sum across absorbed terms remains the fitted absorbed contribution.

After estimation, `predict` can create the covariate index, the fitted value including absorbed fixed effects, the absorbed component, and residuals:

```
. predict double xb, xb
. predict double yhat, xbd
. predict double d_fe, d
. predict double u2, residuals
```

For models with heterogeneous slopes, `predict, d` returns the total absorbed component, including both ordinary fixed effects and heterogeneous slope terms after multiplication by their continuous variables. Likewise, `predict, xbd` returns the full fitted value. The statistics `xbd`, `d`, and `residuals` require that residuals were saved in the original `xhdfe` call. Saved fixed effects are contributions to the fitted value, not category identifiers. As with other absorbed-effects estimators, their level normalization may depend on the design.

4.7 Numerical controls and unsupported cases

The option `tolerancemode()` determines how the convergence tolerance is evaluated. The default, `reghdfe-comparable`, gives `tolerance()` the same relative-update interpretation as in `reghdfe` and is appropriate for tight numerical comparisons. The `xhdfe-fast` mode uses `xhdfe`'s historical, speed-oriented convergence criterion for exploratory runs and speed benchmarks. The `strict-residual` mode applies a more demanding residual-based check and is unavailable with heterogeneous slopes.

For heterogeneous slopes, `convergence(auto)` follows the selected tolerance mode, whereas `convergence(normchange)`, `convergence(reghdfe)`, and `convergence(both)` override the slope-specific stopping rule. Numerical comparisons should therefore report `tolerancemode()` together with `tolerance()`, `absorptionmethod()`, backend, and thread settings.

After absorption, `e(precision_certified)` reports whether the relative normal-equation residual satisfies the numerical limit. With automatic method selection, a solution that meets the iterative stopping rule but fails this check is refined before results are posted.

When a method is forced explicitly, a failed certificate is reported rather than silently replacing the requested method. The certificate remains distinct from `e(converged)`, except in `group()`/`individual()` models, where certification is required for convergence.

The option `dofadjustments()` controls how `xhdfe` counts degrees of freedom lost to absorbed fixed effects. The default, `all`, follows the `reghdfe` convention. Alternatives such as `none`, `firstpair`, `pairwise`, and `clusters` provide sensitivity checks.

The supported `absorb()` syntax includes categorical fixed effects, two-way categorical interactions specified with `#` or `##`, and heterogeneous slopes of the form `fe#c.x` or `fe##c.x`. For categorical fixed effects, `a##b` is treated as the joint fixed effect `a#b`. Users who want both main fixed effects and their interaction should write `a b a#b`. Factor-variable prefixes are supported in ordinary slope variables. Instrumental-variable (2SLS) estimation with absorbed fixed effects is available through the `endogenous()` and `instruments()` options documented in the help file. The two options must be specified together, factor-variable operators are not supported in the IV syntax, and IV is not available in `group()` mode. The benchmarks reported here do not cover IV estimation.

The group-outcome interface follows `reghdfe` syntax, with `group()`, `individual()`, and `aggregation(mean|sum)`. Saving fixed-effect contributions with `savefe` or `savefes()` is not supported when both `group()` and `individual()` are specified. HAC and Driscoll-Kraay standard errors are not implemented.

4.8 Stored results

`xhdfe` is an e-class command. It works with standard postestimation commands such as `estimates store`, `test`, `lincom`, and `nlcom`. Implemented `estat` subcommands include `estat summarize`, `estat vce`, and `estat ic`. Table 4 lists the stored results most relevant for empirical use, validation, and benchmarking.

4.9 Companion commands

This section covers seven companion commands that support estimation, sample preparation, inference, and presentation. The discussion below focuses on the three analytical companions, specifically, `xfe`, `xhdfeakm`, and `xhdfegelbach` which expose or extend the core estimation routines. The remaining four support the associated preparation, inference, and presentation workflows.

`xfe` provides direct access to the absorption routines. It residualizes the listed variables against the absorbed fixed effects and returns the within-transformed variables without estimating a coefficient table. It serves the same role for `xhdfe` that `hdfe` serves for `reghdfe`. Researchers can use the transformed variables in a custom estimator, a bootstrap, or a machine-learning model. Because `xfe` shares the compiled core, the benchmarks describe its principal numerical workload, although the command is not benchmarked separately.

`xhdfeakm` addresses worker-firm applications. It estimates the two-way Abowd et al. (1999) model on the leave-one-out connected set. It reports the variance of the worker and

Table 4: Stored results reported by `xhdfe`

Stored result	Description
Scalars	
`e(N)`	Estimation sample size
`e(N_full)`	Sample size before singleton removal
`e(num_singletons)`	Number of singleton observations dropped
`e(df_a)`	Absorbed degrees of freedom
`e(df_r)`	Residual degrees of freedom
`e(N_clust)`	Number of clusters used for clustered inference
`e(N_clustervars)`	Number of cluster dimensions
`e(N_clust#)`	Number of clusters in cluster dimension #
`e(iterations)`	Absorber iteration count
`e(ic)`	Alias of `e(iterations)` for `estat ic`
`e(converged)`	Absorber convergence indicator
`e(abs_residual)`	Verified absolute normal-equation residual after absorption
`e(abs_residual_rel)`	Verified scale-normalized residual after absorption
`e(precision_certified)`	Indicator that the explicit residual check meets its numerical limit
`e(threads_used)`	Number of backend threads used by the absorber
`e(gpu_used)`	Indicator equal to one when a GPU backend was used
`e(absorption_method_used)`	Absorption method selected by the backend
Macros	
`e(cmd)`	Command name
`e(cmdline)`	Command line as typed
`e(depvar)`	Dependent variable
`e(indepvars)`	Included regressors
`e(absorb)`	Absorbed fixed-effect list as displayed
`e(absvars)`	Absorbed fixed-effect list for compatibility
`e(vce)`	Variance-estimator type
`e(vcetype)`	Variance-estimator label for display
`e(clustvar)`	Cluster variables
`e(resid)`	Residual variable name, when `residuals()` is specified
`e(gpu_backend_requested)`	Requested backend label
`e(gpu_backend)`	Effective backend label
`e(gpu_status)`	Backend status diagnostic
`e(tolerance_mode)`	Absorber stopping mode used by the command
`e(predict)`	Program used for `predict`
`e(estat_cmd)`	Program used for `estat`
Matrices and functions	
`e(b)`	Coefficient vector
`e(V)`	Variance-covariance matrix
`e(sample)`	Estimation-sample indicator

Notes: The table is selective. The help file documents additional stored fit statistics, degrees-of-freedom quantities, group-option macros, and postestimation fields.

firm effects, their covariance, and implied sorting using the biased plug-in decomposition, the homoskedastic correction of Andrews et al. (2008), and the heteroskedasticity-robust leave-out correction of Kline et al. (2020). Leverages can be computed exactly or by Johnson–Lindenstrauss approximation. The command also reports component standard errors and, in the leading-eigenvalue case, Andrews–Mikusheva weak-identification confidence intervals

(Andrews and Mikusheva, 2016).

The third analytical companion, `xhdfegelbach`, addresses coefficient movements between nested specifications. It implements the conditional decomposition of Gelbach (2016), attributing the movement of a base coefficient between a short and a long regression to additive, order-invariant channels while using the same absorption routines.

The analytical companions are deliberately narrow. `xhdfe` remains a `reghdfe`-style high-dimensional fixed-effects estimator; `xfe` exposes its partialling-out routines, while `xhdfeakm` and `xhdfegelbach` add specific decomposition tools rather than a separate modeling framework. The validation and runtime evidence in Section 5 concerns the `xhdfe` regression command, whereas the analytical companions are checked for numerical agreement with their reference implementations rather than benchmarked for speed.

The leave-out point estimates and variance decompositions produced by `xhdfeakm` are checked against Saggio's `LeaveOutTwoWay` (Saggio, 2020), the reference implementation of Kline et al. (2020), and against `pytwoway` where the commands overlap. The Gelbach coefficient decomposition and tested inference results agree with `b1x2` to machine precision. Component standard errors and the Andrews–Mikusheva method are implemented but are not independently revalidated in this paper. Equivalent AKM and Gelbach functionality is available through the Python and R interfaces, and the Python interface can export the leave-out sample to the `pytwoway` format of Lamadon and collaborators' two-way worker-firm toolkit (Bonhomme et al., 2023, Lamadon and Oppenheimer, 2024).

The four supporting commands complete these workflows. `xhdfeconnected` exposes the connected-set preparation used by `xhdfeakm`; `xhdfegelbachbootstrap`, `xhdfegelbachetable`, and `xhdfegelbachcoefplot` provide inference and presentation tools for Gelbach results. Table 5 summarizes these seven commands. Full syntax and options are documented in the command help files.

Table 5: Analytical and supporting companion commands

Command	Purpose
Analytical companions	
`xfe`	Residualize variables against multiple fixed effects without fitting a regression.
`xhdfeakm`	Estimate AKM worker-firm variance decompositions, including plug-in and leave-out corrections.
`xhdfegelbach`	Decompose coefficient changes between short and long regressions into additive, order-invariant channels.
Supporting utilities	
`xhdfeconnected`	Extract the largest leave-one-out connected set used for AKM estimation.
`xhdfegelbachbootstrap`	Compute full-refit bootstrap inference for Gelbach decompositions.
`xhdfegelbachetable`	Format Gelbach results as publication-oriented, multi-panel tables.
`xhdfegelbachcoefplot`	Plot Gelbach contributions as a waterfall chart.

Notes: The installation utility `xhdfegpu` is described separately under backend options.

5 Validation and performance of `xhdfe` on AKM-style regressions

The validation exercise uses two wage regressions in the spirit of Abowd et al. (1999). Both are demanding HDFE specifications that absorb worker, firm, and year effects. The comparisons focus on a small set of wage-profile coefficients and their clustered standard errors.

The data are from Quadros de Pessoal, a restricted-access Portuguese employer census that links workers to firms and records wages, job characteristics, and employment spells. The benchmark extract contains 73,426,387 worker-firm-year observations for 1986–2023. Dropping missing variables common to the two specifications leaves 57,821,050 observations, and singleton removal leaves 55,947,171 estimation observations. Singletons are removed iteratively using the `reghdfe` rule, and all comparisons in this section use the resulting estimation sample. The validation and performance exercises in this section are unweighted.

Let i index workers, j firms, and t years. The dependent variable y_{ijt} is log hourly pay. Following Abowd et al. (1999), both specifications include a scaled experience polynomial as ordinary regressors. Firm-seniority terms enter as common covariates in the first specification and as absorbed firm-specific slopes in the second. Let

$$\mathbf{p}(e_{it}) = \left(e_{it}, e_{it}^2/100, e_{it}^3/1000, e_{it}^4/10000\right)'$$

collect the scaled experience polynomial used in the benchmarks. Let a_{ijt} denote observed firm seniority and b_{ijt} denote the corresponding spline term. The first regression treats seniority as a common covariate while absorbing worker, year, and firm fixed effects:

$$y_{ijt} = \mathbf{p}(e_{it})'\boldsymbol{\beta} + a_{ijt}\delta_1 + b_{ijt}\delta_2 + \underbrace{\theta_i + \lambda_t + \psi_j}_{\text{absorbed fixed effects}} + u_{ijt} \tag{4}$$

The first specification is a standard worker-firm HDFE regression with common returns to firm seniority and standard errors clustered by worker.

The second regression moves the seniority terms into the firm-effect structure. It keeps the same experience polynomial but allows the seniority profile to vary by firm:

$$y_{ijt} = \mathbf{p}(e_{it})'\boldsymbol{\beta} + \underbrace{\theta_i + \lambda_t + \psi_j}_{\text{absorbed fixed effects}} + \underbrace{a_{ijt}\gamma_j + b_{ijt}\eta_j}_{\text{absorbed firm-specific slopes}} + u_{ijt} \tag{5}$$

The second specification absorbs worker and year effects, firm intercepts, and two firm-specific seniority slopes, with standard errors clustered by worker and firm. It is more demanding because the seniority profile is absorbed rather than estimated with common coefficients. The next three subsections use these specifications to assess agreement with `reghdfe`, runtime gains relative to `reghdfe`, and comparisons with alternative implementations of HDFE models.

5.1 Agreement with `reghdfe`

`xhdfe` targets the same linear fixed-effects estimand as `reghdfe`.[6] The relevant numerical question is therefore not whether the two commands produce similar qualitative results, but whether the reported coefficients and standard errors agree up to numerical tolerance. Table 6 reports the maximum absolute differences relative to `reghdfe` for the two AKM-style regressions in (4) and (5). For each regression, the table shows both the CPU and CUDA backends under the current default `tolerancemode(reghdfe-comparable)` convergence rule and under the opt-in `tolerancemode(xhdfe-fast)` rule. All comparisons in this subsection use the default `dofadjustments(all)`, so the standard-error differences also reflect the default absorbed-degree-of-freedom accounting, including cluster-nesting adjustments where relevant.

Table 6: Agreement of `xhdfe` with `reghdfe` on AKM-style regressions

Backend	Tolerance mode	Max abs. coefficient difference	Max abs. SE difference
	Common seniority, equation (4)		
CPU	`reghdfe-comparable`	3.75×10^{-12}	1.45×10^{-13}
CUDA	`reghdfe-comparable`	5.32×10^{-11}	1.50×10^{-12}
CPU	`xhdfe-fast`	2.03×10^{-7}	2.47×10^{-10}
CUDA	`xhdfe-fast`	2.03×10^{-7}	2.39×10^{-10}
	Firm-specific seniority, equation (5)		
CPU	`reghdfe-comparable`	3.08×10^{-10}	5.14×10^{-11}
CUDA	`reghdfe-comparable`	9.61×10^{-9}	1.26×10^{-10}
CPU	`xhdfe-fast`	1.14×10^{-7}	5.83×10^{-9}
CUDA	`xhdfe-fast`	2.18×10^{-8}	1.64×10^{-9}

Notes: Entries are maximum absolute differences across the reported nonconstant coefficients and their standard errors, using `reghdfe` as the reference. The `tolerancemode(reghdfe-comparable)` rows use the current default convergence rule designed for closer comparison with `reghdfe`. `tolerancemode(xhdfe-fast)` uses the historical fast `xhdfe` convergence criterion.

The differences are small in both specifications. Under the fast `xhdfe-fast` mode, the largest coefficient discrepancies are below 3×10^{-7}, and the largest standard-error discrepancies are below 6×10^{-9}. The `reghdfe-comparable` mode tightens coefficient differences to about 10^{-8} or smaller and keeps standard-error differences below 2×10^{-10}. The differences remain small on both CPU and CUDA, showing that backend choice does not materially affect the reported estimates. The `xhdfe-fast` rule trades a small loss in numerical agreement for shorter runtimes, whereas the current default provides closer agreement with `reghdfe`.

[6] We use `reghdfe` as the validation reference because `xhdfe` is designed to reproduce it on overlapping specifications and because it supports the benchmark models. Official Stata HDFE functionality is discussed in the introduction but is not the statistical reference for these comparisons.

5.2 Runtime gains relative to `reghdfe`

This subsection compares `reghdfe` and `xhdfe` on the two AKM-style specifications, using the agreement results in Section 5.1. The first specification absorbs worker, firm, and year effects and estimates common firm-seniority coefficients. The second specification is more demanding because it absorbs firm intercepts and two firm-specific seniority slopes, with standard errors clustered by worker and firm.

Table 7 reports median runtime, iteration counts, and speedups relative to `reghdfe`. The benchmark covers the Stata command and the Python wrapper, both on CPU and CUDA, under two convergence modes. The current default `reghdfe-comparable` mode uses stricter numerical criteria designed to match the reported output of `reghdfe` more closely, while `xhdfe-fast` keeps the historical fast stopping criterion. For eligible standard fixed-effect specifications, the default absorber may select modified LSMR. Heterogeneous-slope specifications instead use sweep-based absorption.

All benchmark timings are estimator-call times measured after data preparation and package loading. The `xhdfe` rows report medians over five execution runs using the backend and tolerance mode shown in the table, with thread settings reported in Appendix A. The `reghdfe` reference uses the same estimation sample, absorbed effects, covariates, clustering choices, and singleton rule, and is measured as one full-sample execution for each specification because each run is much more computationally expensive. Appendix A reports the benchmark environment and timing protocol.

The runtime gains remain large under the stricter `reghdfe-comparable` stopping rule. In the common-seniority specification, the Stata CUDA implementation is 141.4 times faster than the `reghdfe` reference, while the Python CUDA implementation is 209.9 times faster. In the firm-specific-seniority specification, the corresponding speedups are 206.3 and 236.4 times. The `xhdfe-fast` mode shortens the median timed call in all eight backend comparisons. For example, the firm-specific Stata CUDA median falls from 69.5 to 59.5 seconds as the iteration count falls from 151 to 38. Numerical agreement under both rules is documented in Section 5.1.

The interface comparisons are mixed even though both interfaces use the same compiled `xhdfe` core. The benchmark does not isolate the source of these runtime differences. The Python interface times a call on already prepared arrays, whereas the Stata timing also includes plugin transfer, Stata-side variable handling, syntax parsing, result posting, and e-class accounting. Python is faster in all four CUDA comparisons. The observed Stata-to-Python gap is 8.1–8.5 seconds in the common-seniority model and 8.9–9.5 seconds in the firm-specific model. These gaps combine interface and implementation costs and should not be treated as a fixed overhead outside the reported benchmarks.

5.3 Comparison with other implementations

The same two AKM-style specifications also provide a useful comparison across HDFE implementations. Runtime is only one dimension of this comparison: packages also differ in syntax, default numerical methods, variance-covariance conventions, and support for

Table 7: Current `xhdfe` runtimes relative to `reghdfe`

Interface	Backend	Tolerance mode	Iterations	Seconds	Speedup
		Common seniority, equation (4)			
`reghdfe`, Stata	CPU	reference	124	3,667.5	1.0x
`xhdfe`, Python	CPU	`reghdfe-comparable`	24	85.8	42.7x
`xhdfe`, Python	CPU	`xhdfe-fast`	32	53.6	68.4x
`xhdfe`, Python	CUDA	`reghdfe-comparable`	102	17.5	209.9x
`xhdfe`, Python	CUDA	`xhdfe-fast`	32	14.5	252.2x
`xhdfe`, Stata	CPU	`reghdfe-comparable`	24	89.6	41.0x
`xhdfe`, Stata	CPU	`xhdfe-fast`	32	61.3	59.8x
`xhdfe`, Stata	CUDA	`reghdfe-comparable`	102	25.9	141.4x
`xhdfe`, Stata	CUDA	`xhdfe-fast`	32	22.7	161.7x
		Firm-specific seniority, equation (5)			
`reghdfe`, Stata	CPU	reference	249	14,343.0	1.0x
`xhdfe`, Python	CPU	`reghdfe-comparable`	86	195.3	73.4x
`xhdfe`, Python	CPU	`xhdfe-fast`	30	96.7	148.3x
`xhdfe`, Python	CUDA	`reghdfe-comparable`	151	60.7	236.4x
`xhdfe`, Python	CUDA	`xhdfe-fast`	38	49.9	287.2x
`xhdfe`, Stata	CPU	`reghdfe-comparable`	86	201.6	71.2x
`xhdfe`, Stata	CPU	`xhdfe-fast`	30	111.6	128.5x
`xhdfe`, Stata	CUDA	`reghdfe-comparable`	151	69.5	206.3x
`xhdfe`, Stata	CUDA	`xhdfe-fast`	38	59.5	241.2x

Notes: Seconds are estimator-call times. `xhdfe` rows report medians over five execution runs. The `reghdfe` rows are one full-sample execution for each specification, measured under the same sample, covariates, absorbed effects, clustering choices, and singleton rule. Speedup is the `reghdfe` reference time divided by the `xhdfe` time within the same specification, computed from unrounded timings. `xhdfe` runs use 16 threads. Iteration counts are method-specific diagnostics and should not be read as identical computational units across absorbers. Appendix A reports the benchmark environment and timing protocol.

heterogeneous fixed-effect slopes. The comparison therefore reports both execution time and agreement with the `reghdfe` benchmarks.

Table 8 includes the `reghdfe` reference estimates, the `xhdfe` timings from Table 7, and three alternative implementations: `fixest` 0.14.2 (Bergé et al., 2026) under R 4.6.0, `FixedEffectModels.jl` 1.13.2 (Gomez and contributors, 2026) under Julia 1.12.1 with `Vcov.jl` 0.8.3, and `pyfixest` 0.60.0 (The PyFixest Authors, 2025).[7] All timings in Table 8 use the same benchmark input sample as Sections 5.1 and 5.2.

For the common-seniority specification, all three alternative implementations successfully

[7]The respective package repositories can be found at:

- `fixest` in `https://github.com/lrberge/fixest`,
- `FixedEffectModels.jl` in `https://github.com/FixedEffects/FixedEffectModels.jl`,
- `Vcov.jl` in `https://github.com/FixedEffects/Vcov.jl`, and
- `pyfixest` in `https://github.com/py-econometrics/pyfixest`.

Table 8: HDFE implementations on the AKM-style regressions

Implementation	Version	Backend	Coef. diff.	SE diff.	Seconds	Speedup
Common seniority, equation (4)						
`reghdfe`, Stata	6.12.5	CPU	0	0	3,667.5	1.0x
`FixedEffectModels.jl`	1.13.2	CPU	1.30e-10	3.96e-12	916.3	4.0x
`fixest`	0.14.2	CPU	6.56e-10	4.48e-12	486.5	7.5x
`pyfixest`	0.60.0	CPU	5.81e-12	2.08e-13	110.4	33.2x
`FixedEffectModels.jl`	1.13.2	CUDA	1.32e-10	5.39e-12	105.7	34.7x
`xhdfe`, Stata `reghdfe-comparable`	2.23.1	CPU	3.75e-12	1.45e-13	89.6	41.0x
`xhdfe`, Python `reghdfe-comparable`	2.23.1	CPU	3.69e-12	1.43e-13	85.8	42.7x
`xhdfe`, Stata `xhdfe-fast`	2.23.1	CPU	2.03e-07	2.47e-10	61.3	59.8x
`xhdfe`, Python `xhdfe-fast`	2.23.1	CPU	2.03e-07	2.47e-10	53.6	68.4x
`xhdfe`, Stata `reghdfe-comparable`	2.23.1	CUDA	5.32e-11	1.50e-12	25.9	141.4x
`xhdfe`, Stata `xhdfe-fast`	2.23.1	CUDA	2.03e-07	2.39e-10	22.7	161.7x
`xhdfe`, Python `reghdfe-comparable`	2.23.1	CUDA	5.34e-11	1.50e-12	17.5	209.9x
`xhdfe`, Python `xhdfe-fast`	2.23.1	CUDA	2.03e-07	2.39e-10	14.5	252.2x
Firm-specific seniority, equation (5)						
`reghdfe`, Stata	6.12.5	CPU	0	0	14,343.0	1.0x
`pyfixest`	0.60.0	CPU	–	–	–	–
`FixedEffectModels.jl`	1.13.2	CPU	9.50e-11	2.08e-05	2,776.1	5.2x
`fixest`	0.14.2	CPU	3.43e-09	1.91e-05	1,370.7	10.5x
`xhdfe`, Stata `reghdfe-comparable`	2.23.1	CPU	3.08e-10	5.14e-11	201.6	71.2x
`xhdfe`, Python `reghdfe-comparable`	2.23.1	CPU	3.08e-10	5.14e-11	195.3	73.4x
`FixedEffectModels.jl`	1.13.2	CUDA	1.03e-10	2.08e-05	169.7	84.5x
`xhdfe`, Stata `xhdfe-fast`	2.23.1	CPU	1.14e-07	5.83e-09	111.6	128.5x
`xhdfe`, Python `xhdfe-fast`	2.23.1	CPU	1.14e-07	5.83e-09	96.7	148.3x
`xhdfe`, Stata `reghdfe-comparable`	2.23.1	CUDA	9.61e-09	1.26e-10	69.5	206.3x
`xhdfe`, Python `reghdfe-comparable`	2.23.1	CUDA	9.61e-09	1.26e-10	60.7	236.4x
`xhdfe`, Stata `xhdfe-fast`	2.23.1	CUDA	2.18e-08	1.64e-09	59.5	241.2x
`xhdfe`, Python `xhdfe-fast`	2.23.1	CUDA	2.18e-08	1.64e-09	49.9	287.2x

Notes: The `xhdfe` rows report version 2.23.1 median runtimes from Table 7. Comparator rows come from earlier benchmark runs under the same protocol and were not rerun in this release refresh. Non-`reghdfe` entries report medians over five runs; each `reghdfe` reference is one full-sample execution. Each `FixedEffectModels.jl` trial starts a fresh Julia process, so estimator-triggered compilation or JIT work is included in every timing; the other five-run implementations repeat calls within one process. Speedup is the `reghdfe` reference time divided by the entry runtime within the same specification. Accuracy gaps are maximum absolute differences relative to `reghdfe`, computed over six nonconstant coefficients in the common-seniority specification and four in the firm-specific-seniority specification. Rows with reported speedups are ordered by speedup within each panel; the unsupported `pyfixest` row follows the `reghdfe` reference. Standard-error gaps in the two-way-clustered firm-specific-seniority specification reflect both numerical differences and package-specific variance-covariance conventions. The blank `pyfixest` entries indicate that the program does not support the heterogeneous-slope syntax used in this specification.

estimate the model and agree closely with `reghdfe`. The largest coefficient difference is 6.56×10^{-10}, and the largest standard-error difference is 5.39×10^{-12}. `xhdfe` records the shortest runtimes among the implementations and configurations reported in the table. Its CUDA configurations are faster than their CPU counterparts. The fastest `xhdfe` CUDA configuration completes the common-seniority regression in 14.5 seconds. The closest non-`xhdfe` runtime in this panel is `FixedEffectModels.jl` on CUDA, at 105.7 seconds. Even so, both `xhdfe-fast` CPU configurations are faster than that CUDA alternative.

Among CPU-only non-`xhdfe` alternatives, `pyfixest` is fastest, with a median runtime of 110.4 seconds. Under the `reghdfe-comparable` tolerance mode, `xhdfe` takes 89.6 seconds through Stata and 85.8 seconds through Python. The Python comparison is the more direct one because both commands are invoked from Python. Relative to `pyfixest`, `xhdfe` is 19% faster through Stata and 22% faster through Python. Under `xhdfe-fast`, the Python-to-Python ratio rises to 2.06.

The firm-specific-seniority specification is more informative about feature coverage. `fixest` and `FixedEffectModels.jl` estimate the model, and their coefficient gaps relative to `reghdfe` remain small. Their clustered standard errors differ by about 2×10^{-5}. The benchmark does not isolate how much of these gaps comes from numerical differences versus package-specific variance-covariance conventions. In this harder specification, the shortest non-`xhdfe` runtime is again the CUDA configuration of `FixedEffectModels.jl`. It is still slower than several `xhdfe` configurations, including Python CPU with `xhdfe-fast`. `pyfixest`, by contrast, does not support the heterogeneous fixed-effect syntax used in this specification. For this reason, it is listed as unsupported for this benchmark rather than assigned an accuracy or runtime result.

6 Conclusion

`xhdfe` is intended to solve a practical problem rather than introduce a new econometric estimator. Applied researchers already rely on `reghdfe`-style estimators to fit linear models with many absorbed fixed effects. The contribution of `xhdfe` is to preserve that familiar Stata interface for an important subset of linear HDFE models while moving the expensive absorption step into a compiled C++ backend with CPU threading and optional CUDA execution.

The estimator analysis shows that `xhdfe` targets the same partialled-out least-squares problem as `reghdfe` on the overlapping specifications studied here. The syntax section documents the command and its options, including multiple absorbed fixed effects, clustered and robust standard errors, heterogeneous slopes in tested cases, backend choices, saved fixed effects, residuals, fitted values, and stored results. The AKM-style application then shows that `xhdfe` reproduces `reghdfe` coefficients and standard errors within tight numerical tolerances while delivering large runtime gains. Under the current default `reghdfe-comparable` mode, the Stata command is 41.0–71.2 times faster than the `reghdfe` references on CPU and 141.4–206.3 times faster on CUDA. Under the speed-oriented `xhdfe-fast` mode, the corresponding Stata ranges are about 60–128 times on CPU and 162–241 times on CUDA. The Python wrapper is faster in all four paired CUDA comparisons, although its timed call excludes several Stata-side tasks.

The comparison with other HDFE implementations places these gains in context. The closest non-`xhdfe` runtime comes from `FixedEffectModels.jl` on CUDA, although several CPU `xhdfe` configurations are faster. The same comparison also shows why speed is not sufficient on its own. Feature coverage, variance-covariance conventions, syntax support, and agreement with the reference Stata implementation all matter when deciding whether an

implementation is a substitute for a particular empirical task.

The development history also points to a broader use for agentic AI in empirical software. In this project, these tools assisted with implementation, refactoring, and code review, while established statistical results and reproducible benchmarks provided independent checks on the changes. Similar tools could reduce the cost of improving other computationally intensive Stata commands without changing their underlying estimators.

`xhdfe` does not yet cover every `reghdfe` feature. The paper documents the command as it currently stands and validates the functionality used in the reported benchmarks. Future extensions should be evaluated through numerical agreement with established implementations and reproducible runtime evidence.

Acknowledgments

`xhdfe` is a reimplementation designed to reproduce results from and interoperate with prior high-dimensional fixed-effects software. We acknowledge `reghdfe` by Sergio Correia (Stata), `fixest` by Laurent Bergé (R), `pyfixest` by Alexander Fischer and collaborators (Python), and `FixedEffectModels.jl` by Matthieu Gomez and collaborators (Julia). The companion commands build on and are validated against `LeaveOutTwoWay` by Raffaele Saggio, `b1x2` by Jonah Gelbach, and `pytwoway` by Thibaut Lamadon and Adam A. Oppenheimer.

We thank Paulo Guimarães (Banco de Portugal), Marta Silva (Banco de Portugal), and Nelson Areal (EEG / University of Minho) for helpful discussions. We especially thank Sergio Correia for feedback on benchmarking, tolerances, and `reghdfe`-comparable validation. We also thank Alexander Fischer for sharing updates on the graph-based fixed-effects demeaning strategy he is developing with Kristof Schröder: a modified LSMR solver with an additive-Schwarz preconditioner built on the worker-firm bipartite graph (the `within` project, `https://github.com/py-econometrics/within`), which has been helpful for our ongoing work on high-dimensional demeaning. Nelson Areal and Miguel Portela presented "Parallel and Cross-Language Computing: A Hands-On Workshop for Empirical Researchers" at BPLIM's workshop *Speeding Up Empirical Research: Tools and Techniques for Fast Computing* (BPLIM2025), where an earlier version of this work was shared. We also thank Universidade do Minho, Banco de Portugal, and FCT (Portuguese Foundation for Science and Technology, UID/03182/2025) for financial support. The usual disclaimer applies.

References

Abowd, John M., Francis Kramarz, and David N. Margolis (1999) "High Wage Workers and High Wage Firms," *Econometrica*, 67, 251–333, 10.1111/1468-0262.00020.

Andrews, Isaiah and Anna Mikusheva (2016) "A Geometric Approach to Nonlinear Econometric Models," *Econometrica*, 84, 1249–1264, 10.3982/ECTA12030.

Andrews, Martyn J., Len Gill, Thorsten Schank, and Richard Upward (2008) "High Wage

Workers and Low Wage Firms: Negative Assortative Matching or Limited Mobility Bias?" *Journal of the Royal Statistical Society: Series A*, 171, 673–697, 10.1111/j.1467-985X.2007.00533.x.

Bergé, Laurent R., Kyle Butts, and Grant McDermott (2026) "fixest: A Fast and Feature-Rich Framework for Econometric Estimations in R," *arXiv preprint arXiv:2601.21749*, 10.48550/arXiv.2601.21749, Benchmarked development build of R package version 0.14.2 (archived runs 18 Jun 2026). CRAN 0.14.2 published 2026-06-26. doi: `https://doi.org/10.48550/arXiv.2601.21749`.

Bonhomme, Stéphane, Kerstin Holzheu, Thibaut Lamadon, Elena Manresa, Magne Mogstad, and Bradley Setzler (2023) "How Much Should We Trust Estimates of Firm Effects and Worker Sorting?" *Journal of Labor Economics*, 41, 291–322, 10.1086/720009.

Carneiro, Anabela, Paulo Guimarães, and Pedro Portugal (2012) "Real Wages and the Business Cycle: Accounting for Worker, Firm, and Job Title Heterogeneity," *American Economic Journal: Macroeconomics*, 4, 133–152, 10.1257/mac.4.2.133.

Correia, Sergio (2017) "Linear Models with High-Dimensional Fixed Effects: An Efficient and Feasible Estimator," Working Paper, Available at `http://scorreia.com/research/hdfe.pdf`.

Correia, Sergio and Noah Constantine (2014) "REGHDFE: Stata Module to Perform Linear or Instrumental-Variable Regression Absorbing Any Number of High-Dimensional Fixed Effects," Statistical Software Components S457874, Boston College Department of Economics, Version 6.13.1, 10 Jan 2026. RePEc revision 11 Jan 2026. Available at `https://ideas.repec.org/c/boc/bocode/s457874.html`.

Fischer, Alexander and Kristof Schröder (2026) "Graph Preconditioning for High-Dimensional Fixed Effects Regression," Mimeo.

Fong, David Chin-Lung and Michael A. Saunders (2011) "LSMR: An Iterative Algorithm for Sparse Least-Squares Problems," *SIAM Journal on Scientific Computing*, 33, 2950–2971, 10.1137/10079687X.

Gelbach, Jonah B. (2016) "When Do Covariates Matter? And Which Ones, and How Much?" *Journal of Labor Economics*, 34, 509–543, 10.1086/683668.

Gomez, Matthieu and contributors (2026) "FixedEffectModels.jl: Fast Estimation of Linear Models with High Dimensional Categorical Factors," `https://github.com/FixedEffects/FixedEffectModels.jl`, Julia package repository, version 1.13.2, released 2026-06-05. Accessed 2026-07-06.

Guimarães, Paulo and Pedro Portugal (2010) "A Simple Feasible Procedure to Fit Models with High-Dimensional Fixed Effects," *The Stata Journal*, 10, 628–649, 10.1177/1536867X1101000406.

Kline, Patrick, Raffaele Saggio, and Mikkel Sølvsten (2020) "Leave-Out Estimation of Variance Components," *Econometrica*, 88, 1859–1898, 10.3982/ECTA16410.

Lamadon, Thibaut and Adam A. Oppenheimer (2024) "pytwoway: Two-Way Fixed Effect Models for Worker-Firm Data in Python," `https://github.com/tlamadon/pytwoway`, Python package repository. Accessed 2026-07-06.

Paige, Christopher C. and Michael A. Saunders (1982) "LSQR: An Algorithm for Sparse Linear Equations and Sparse Least Squares," *ACM Transactions on Mathematical Software*, 8, 43–71, 10.1145/355984.355989.

Portugal, Pedro, Hugo Reis, Paulo Guimarães, and Ana Rute Cardoso (2026) "What Lies behind the Returns to Schooling: The Role of Labor Market Sorting and Worker Heterogeneity," *The Review of Economics and Statistics*, 10.1162/rest_a_01482, Forthcoming.

Saad, Yousef (2003) *Iterative Methods for Sparse Linear Systems*, Philadelphia, PA: Society for Industrial and Applied Mathematics, 2nd edition, 10.1137/1.9780898718003.

Saggio, Raffaele (2020) "LeaveOutTwoWay: Leave-Out Estimation of Variance Components," `https://github.com/rsaggio87/LeaveOutTwoWay`, MATLAB package. Reference implementation of Kline et al. (2020). Accessed 2026-07-06.

StataCorp (2026) "High-dimensional fixed effects," `https://www.stata.com/new-in-stata/high-dimensional-fixed-effects/`, Official Stata documentation, accessed 2026-05-21.

The PyFixest Authors (2025) "pyfixest: Fast High-Dimensional Fixed Effect Estimation in Python," `https://github.com/py-econometrics/pyfixest`, Python package repository, version 0.60.0. Accessed 2026-07-06.

Appendix

A Benchmark environment and timing protocol

The AKM benchmark comparisons were executed on an institutional server. Table A1 reports the hardware and software environment used for the comparisons in Section 5. The benchmarked `xhdfe` implementation is the same across the Stata command and Python wrapper. Both interfaces call the same compiled core.

Table A1: Benchmark environment for AKM runtime comparisons

Item	Benchmark setting
Server and Stata	Institutional server running StataNow/MP 19.5 with a 16-core license on Linux x86-64
CPU	Intel Xeon Gold 6542Y with 24 physical cores and 48 logical CPUs. Stata set to 16 processors
Memory and GPU	1006.74 GB usable memory reported by Stata and an NVIDIA H100 NVL GPU
CUDA and compiler	CUDA 12.8.93, NVIDIA driver 570.195.03, GCC/G++ 11.5.0, and OpenMP 4.5
`xhdfe`/`xfe`	`xhdfe` 2.23.1 and `xfe` 1.11.0 (6 Aug 2026), benchmarked at release commit `d9958df3`. The Stata commands and Python wrapper call the same compiled core
Stata comparator	`reghdfe` 6.12.5 (27 Dec 2023) with `ftools` 2.49.1 (08 Aug 2023). The current `reghdfe` release is 6.13.1 (10 Jan 2026). Runs use `tolerance(1e-8)` and `iterate(10000)`
`fixest`	R 4.6.0 with `fixest` 0.14.2 (development build benchmarked 18 Jun 2026; CRAN 0.14.2 published 26 Jun 2026)
`FixedEffectModels.jl`	Julia 1.12.1 with `FixedEffectModels.jl` 1.13.2 (5 Jun 2026) and `Vcov.jl` 0.8.3
`pyfixest`	Python 3.12.12 with `pyfixest` 0.60.0

Notes: The environment describes the server used for the AKM comparisons reported in Tables 6–8. The three external comparator programs use a fixed-effect demeaning tolerance of 10^{-8} and a 10,000-iteration cap, matching the `reghdfe` settings. For `pyfixest`, these settings apply to `fixef_atol` and `fixef_btol`.

Timing begins after the benchmark data have been loaded. Stata `xhdfe` timings include Stata-side work and plugin execution: singleton removal, absorption, covariance computation, and result posting. All programs exclude data import and package loading; the `xhdfe` timings also exclude plugin compilation. No preliminary estimator call is discarded as a warm-up. The Stata and Python `xhdfe` interfaces, `fixest`, and `pyfixest` repeat calls within one process. Each `FixedEffectModels.jl` trial starts a fresh Julia process, so estimator-triggered compilation or JIT work is included in every Julia timing. The benchmark therefore measures estimator calls on prepared data rather than end-to-end data import.

Tables 7 and 8 report five-run medians for each non-`reghdfe` configuration. Each `reghdfe` reference is one full-sample run: 3,667.5 seconds for common seniority and 14,343.0 seconds for firm-specific seniority. Because each speedup uses a single `reghdfe` timing in the numerator, the reported ratios retain any run-to-run variation in that reference timing.