ZEUS: An Efficient GPU Optimization Method Integrating PSO, BFGS, and Automatic Differentiation

Dominik Soós*, Marc Paterno*†, Desh Ranjan*, Mohammad Zubair*
*Department of Computer Science
Old Dominion University
Norfolk, Virginia, USA
†Fermi National Accelerator Laboratory
Batavia, Illinois, USA

***Abstract*—We introduce a novel, efficient computational method, ZEUS, for numerical optimization, and provide an open-source implementation. It has four key ingredients: (1) particle swarm optimization (PSO), (2) the use of the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method, (3) automatic differentiation (AD), and (4) GPUs. Our approach addresses the computational challenges inherent in high-dimensional, non-convex optimization problems. In the first phase of the algorithm, we get a potentially good set of starting points using PSO. Thereafter, we run BFGS independently in parallel from these starting points. BFGS is one of the best-performing algorithms for numerical optimization. However, it requires the gradient of the function being optimized. ZEUS integrates automatic differentiation into BFGS thus avoiding the need for the user to calculate derivatives explicitly. The use of GPUs allows ZEUS to speed up the calculations substantially. We carry out systematic studies to explore the trade-offs between the number of PSO iterations taken, starting points, and BFGS iteration depth. We show that a handful of iterations of PSO can improve global convergence when combined with BFGS. We also present performance studies using common test functions. The source code can be found at https://github.com/fnal-numerics/global-optimizer-gpu.**

***Index Terms*—numerical optimization, parallel computing, swarm intelligence, automatic differentiation**

I. INTRODUCTION

A wide range of domains make use of numerical optimization, including particle physics simulations, machine learning, and financial modeling. Multidimensional non-convex global optimization can be difficult due to a number of reasons. One of these reasons is that the solution space grows exponentially with the dimensionality of the function being optimized.

The classical sequential numerical optimization algorithms start from an initial "guess" and then use the function gradient to move in the direction that reduces the value of the objective function. However, such conventional optimization techniques often fail to navigate complex landscapes with many local minima or narrow valleys where the gradient is close to zero. Purely gradient-based methods like stochastic gradient descent [Ketkar(2017)], mini-batching [Li et al.(2014)], [Singh et al.(2024)], and stochastic variance-reduced gradient [Reddi et al.(2016)] get stuck in flat regions or local minima when dealing with non-convex landscapes.

To tackle difficult landscapes with many local minima, the particle swarm optimization (PSO) method has been widely adopted for its ability to handle global searches by exploring multiple regions of the hyperspace. However, PSO by itself struggles with problems with flat regions.

The use of graphical processing units (GPUs) has become popular in parallel computing, as they can run thousands of computations simultaneously. Parallel computing approaches have been proposed that distribute the computations of the gradients [Zinkevich et al.(2010)].

This paper investigates how parallel computing on GPUs can improve non-convex global optimization strategies. We propose a novel, GPU-based global optimization algorithm named ZEUS that combines the particle swarm optimization (PSO) with popular quasi-Newton optimization (BFGS) [Broyden(1970)], [Fletcher(1970)], [Goldfarb(1970)], [Shanno(1970)]. Additionally, the method provides a built-in forward-mode automatic differentiation (AD) library, thus avoiding the explicit calculation of gradients needed in the BFGS method. We propose an approach that initializes the optimization from a multitude of random starting points in multidimensional solution space, utilizing GPUs to concurrently execute the BFGS optimization algorithm using forward-mode AD. Our contributions are summarized below.

- We provide a parallel multistart optimization algorithm with automatic differentiation that combines the advantages of the PSO and BFGS methods as well as automatic computation of AD for accuracy and ease of use.
- Our CUDA C++ implementation of the algorithm provides a 10- to 100-fold speedup compared to our serial

implementation.
- We provide insight into the trade-off between the number of PSO steps taken and the number of optimizations run concurrently through our extensive studies.

Our extensive experiments not only confirm that a handful of PSO iterations can boost global convergence rates, as shown in Figure 3, but also reveal hyperparameter trade-offs for different functions. We further discuss limitations like the failure condition on functions like the Ackley function with discontinuous gradients. We outline future plans to reduce computational complexity in the parts of the algorithm that dominate runtime.

The rest of the paper is organized as follows: In the next section, we present some background and previous related work. In Section III, we present the sequential ZEUS algorithm. Section IV presents the parallel version of the algorithm. Section V presents our experiments and results. The last section discusses our results and observations as well as future work. In the next section, we present the background information and the related work to our approach, where we discuss each component of the algorithm, including multistart and swarm-based algorithms, BFGS with AD, and GPUs.

II. BACKGROUND AND RELATED WORK

Stochastic multistart and evolutionary algorithms, where the optimizer starts from many points, are not a new concept. The earliest strategies date back to the 1970s, the first multistart method, which proposed the idea of launching many optimizations from many different points to improve the chance of finding the global optimum [Goldstein and Price(1971)], [Boender et al.(1982)] and was later implemented as GLOBAL [Csendes(1988)].

The PSO method is widely used in the literature to improve global convergence. [Li et al.(2011)] combined PSO and BFGS, but run on a CPU and do not leverage automatic differentiation or parallel GPUs. Other approaches used a two-phase global-local scheme, but neither used PSO nor BFGS, nor AD [Ferreiro et al.(2019)]. Similarly, [Barkalov and Gergel(2016)] implemented a GPU-based global search by space-filling, but no PSO, BFGS, or AD was used. Our unique combination is the first C++/CUDA-based library to integrate:

- global search with PSO
- local refinement via BFGS
- automatic differentiation for gradient calculation
- massively parallel GPU

The well-established BFGS algorithm that originates from the works of Broyden [Broyden(1970)], Fletcher [Fletcher(1970)], Goldfarb [Goldfarb(1970)], and Shanno [Shanno(1970)] uses quasi-Newton updates to approximate the Hessian and typically converges in fewer iterations than first-order methods. [Pu and Yu(1990)] showed that BFGS combined with a Wolfe line search enjoys global convergence properties even in moderately high dimensions, further motivating its application to non-convex problems.

A major challenge with gradient-based methods, however, is the need for accurate derivative information. Manual derivation is error-prone and often impractical for complex, high-dimensional functions. This is where AD becomes invaluable.

A. Automatic Differentiation

In recent years, AD has gained widespread attention, not only in machine learning but also in scientific computing [Hueckelheim et al.(2023)], [Baydin et al.(2018)], [Bücker(2006)], [Margossian(2019)]. AD is attractive because it provides more accurate derivatives with minimal overhead without the need for the user to provide manual derivation of the gradient of the objective function. For instance, [Zubair et al.(2023)] implemented a forward-mode AD technique on GPUs for computational fluid dynamics, achieving performance close to the hardware's peak throughput with higher accuracy than manual derivative calculation. Likewise, [Grabner et al.(2008)] used AD for 2D/3D registration on GPUs, addressing challenges in medical imaging.

Recent advances have further accelerated AD on GPUs using modern compiler techniques. Google Brain's JAX framework [Bradbury et al.(2018)] offers a user-friendly, NumPy-like interface that supports just-in-time, or JIT, compilation and GPU acceleration while providing robust automatic differentiation. However, their implementation is not accessible within user-defined GPU kernels. Despite these advances, most available AD libraries, such as those in DLib by [King(2009)] or the Stan Math Library [Carpenter et al.(2015)], still predominantly target CPU execution.

B. Swarm-Intelligence Algorithms

Another type of multistart algorithm related to our work is Particle Swarm Optimization (PSO), first proposed by Kennedy and Eberhart [Kennedy and Eberhart(1995)] then later improved by Shi and Eberhart [Shi and Eberhart(1998)] and further stabilized by Clerc and Kennedy [Clerc and Kennedy(2002)]. It was introduced as a population-based stochastic optimization technique inspired by social behaviors in animals. PSO algorithms use many particles that move toward promising regions in the search space based on information shared between the particles [Kennedy and Eberhart(1995)].

Different variants of the PSO algorithm have been developed [Jain et al.(2022)] that utilize AD [Della Santa(2024)], [Noel(2012)], [Thobirin and Yanto(2015)], or BFGS [Li et al.(2011)], [Wu et al.(2014)], [Zhang et al.(2016)], [Nezhad et al.(2013)], or LBFGS using GPUs [Dixit et al.(2024)], but not all four together. While the PSO approach attempts to converge to a global minimum, each particle is influenced by the others. It does not guarantee global convergence. Instead of each optimization moving to a single point as a swarm, we explore the search space while also adjusting the velocity of each particle based on the global best. Although multistart and swarm-based techniques have been around for a long time, GPU-accelerated AD and quasi-Newton methods like BFGS have not been used together much. Our approach is designed to handle complex non-convex landscapes like Rastrigin and

Rosenbrock functions by running many separate optimizations at the same time from a variety of random starting points. We only consider GPU-accelerated algorithms for comparison.

C. GPU-Accelerator

ParallelParticleSwarms [Dixit et al.(2024)] is the most related work to our work since they combine GPU with BFGS and SciML allows for AD computation. However, after extensive communication with the authors, we were unable to use their hybrid method.

While multistart and swarm-based algorithms have been extensively studied, and recent advancements have accelerated automatic differentiation on GPUs using modern compiler techniques such as Enzyme [Moses et al.(2021)], a comprehensive integration of PSO, forward-mode AD with the BFGS algorithm on GPUs remains under-explored. Our work addresses this gap by concurrently executing multiple GPU-accelerated BFGS optimization threads initialized by PSO, leveraging forward-mode AD. This approach not only enhances computational efficiency, but also improves the likelihood of converging to the global minimum. In the next section, we provide an overview of the details of the sequential ZEUS algorithm.

III. SEQUENTIAL ZEUS ALGORITHM

The main methodology implements a hybrid PSO with a gradient-based BFGS using AD for the gradient calculation at each iteration. Algorithm 1 describes the sequential version. The bold variables indicate that they are vectors of size $N \times dim$ or dim, where N is the number of particles, and dim is the dimension of the objective function. The algorithm can be broken into two main phases.

In the first phase, using Algorithm 2 we initialize the $\mathbf{swarm}$ of size N, where $\mathbf{swarm}$ stores the current coordinates for each particle for each dimension. We assign $\mathbf{V}$ to each particle to hold the velocities in each dimension. Then, using Algorithm 3 we update the swarm using each particle's best position $\mathbf{pX}$ with its best value pB and the global best gB for each iteration.

The second phase is Algorithm 4, which is the sequential BFGS. We introduce a convergence criterion $required_c$, which holds the number of required convergences set by the user. We use Algorithm 5 to calculate the derivatives for each dimension using our automatic differentiation library. Then, we utilize a backtracking line search method by [Armijo(1966)] described in Algorithm 6, to calculate an optimal step size α. The algorithm will keep looping until we have enough optimizations that have converged to the threshold Θ set by the user. If an optimization does not reach an area where the norm of the gradient is sufficiently small in fewer than $iter_{bfgs}$ iterations, then we terminate that optimization with failure status. The following sections provide an overview of each ingredient of our method.

A. Specifying the starting points

Optimization performances vary greatly depending on the initialization of the algorithm. Therefore, it is beneficial to explore methods other than random number generation to add intelligence to the algorithm.

Algorithm 1 Sequential PSO-BFGS

function SEQUENTIALZEUS($f, N, \text{range}, \text{iter}_{pso}, \text{iter}_{bfgs}, \Theta, required_c, w,\ c_1,\ c_2$)
  Allocate $\mathbf{swarm}[N * dim], \mathbf{V}[N * dim], \mathbf{pX}[N * dim], \mathbf{pB}[N], \mathbf{gX}[dim]$
  $gB \leftarrow +\infty$
  $\mathbf{swarm} \leftarrow$INITSWARM($f, N, \text{range}, \mathbf{swarm}, \mathbf{V}, \mathbf{pX}$,)
  $c \leftarrow 0$; ▷ number of converged runs
  for $j \leftarrow 0$ to $\text{iter}_{pso} - 1$ **do**
    $\mathbf{swarm} \leftarrow$UPDATESWARM($f, N, \text{range}, \mathbf{swarm}, \mathbf{V}, \mathbf{pX}, pVal, \mathbf{gX}, gF, w,\ c_1,\ c_2$)
  end for
  for all $i \leftarrow 0, \ldots, N-1$ **do**
    $r \leftarrow$ SERIALBFGS($f, \text{range}, \text{iter}_{bfgs}, \mathbf{swarm}[\mathbf{i}], \Theta$)
    if $r.fval < gF$ **then**
      $gBs \leftarrow r$
    end if
    if $r.\text{status} = 1$ **then**
      $c \leftarrow c + 1$
      if $c = \text{required}_c$ **then**
        break ▷ stop early once enough runs have converged
      end if
    end if
  end for
  return estimated global minimum gB
end function

1) Random Number Generation: For the sequential random number generation, we relied on standard libraries random device and their uniform distribution. However, starting from random points is suboptimal and can be improved using PSO.

2) Randomness improved by PSO: We developed the serial version of this algorithm that the sequential ZEUS algorithm is using. We integrate PSO into the ZEUS algorithm as an option to initialize the starting points using random numbers and then use swarm intelligence to improve the location of such starting points. The initialization of the swarm is being done in Algorithm 2. Then, we update the swarm using Algorithm 3. Hyperparameter optimization could be its own research project. In this paper we have used these hyperparameters for the PSO: $w = 0.5, c_1 = 1.2, c_2 = 1.5$, where w is the inertia, c_1 is the cognitive coefficient, and c_2 is the social coefficient. Previous work has used the same parameters in their implementation [Deboucha et al.(2020)]. Once the starting points have been specified, we continue to BFGS.

B. BFGS

The BFGS [Broyden(1970)], [Fletcher(1970)], [Goldfarb(1970)], [Shanno(1970)] algorithm is a well-known quasi-Newton method that approximates the inverse of the Hessian matrix using a combination of rank-1 updates shown in Algorithm 4. Algorithm 4 also summarizes the exact update that is used to calculate H_{k+1} at each iteration. We calculate

Algorithm 2 Initialization of the swarm

function INITSWARM($f, N, range, \mathbf{swarm}, \mathbf{V}, \mathbf{pX}, pVal, \mathbf{gX}, gB$)
  $lower \leftarrow$ range.lower, $upper \leftarrow$ range.upper
  $vel\_range \leftarrow (upper - lower)$
  for $i \leftarrow 0$ **to** $N - 1$ **do**
    $\mathbf{swarm[i]} \leftarrow \text{UniformSample}(lower, upper)$
    $\mathbf{V[i]} \leftarrow \text{UniformSample}(-vel\_range, vel\_range)$
    $\mathbf{pX[i]} \leftarrow \mathbf{swarm[i]}$
    $pVal[i] \leftarrow f(\mathbf{swarm[i]})$
    if $i = 0$ **or** $pVal[i] < gF$ **then**
      $gF \leftarrow pVal[i]$
      $\mathbf{gX} \leftarrow \mathbf{pX[i]}$
    end if
  end forreturn $swarm$
end function

Algorithm 3 Sequential function to update each particle's velocity, position, and bests

function UPDATESWARM($f, N, \mathbf{swarm}, \mathbf{V}, \mathbf{pX}, pVal, \mathbf{gX}, gB, w, c_1, c_2$)
  for $i \leftarrow 0$ **to** $N - 1$ **do**
    $\mathbf{r_1} \leftarrow \text{UniformSample}(0, 1)$
    $\mathbf{r_2} \leftarrow \text{UniformSample}(0, 1)$
    $\mathbf{x} \leftarrow \mathbf{swarm[i]}, \mathbf{v} \leftarrow \mathbf{V[i]}, \mathbf{p} \leftarrow \mathbf{pX[i]}$
    $\mathbf{g} \leftarrow \mathbf{gX}$
    $\mathbf{v}' \leftarrow w\,\mathbf{v} + c_1\,\mathbf{r_1}\,(\mathbf{p} - \mathbf{x}) \; + \; c_2\,\mathbf{r_2}\,(\mathbf{g} - \mathbf{x})$
    $\mathbf{x}' \leftarrow \mathbf{x} + \mathbf{v}'$
    $\mathbf{V[i]} \leftarrow \mathbf{v}', \quad \mathbf{swarm[i]} \leftarrow \mathbf{x}'$
    $fval \leftarrow f(\mathbf{swarm[i]})$
    if $fval < pVal[i]$ **then** ▷ update personal best
      $pVal[i] \leftarrow fval$
      $\mathbf{pX[i]} \leftarrow \mathbf{swarm[i]}$
    end if
    if $fval < gF$ **then** ▷ update global best
      $gF \leftarrow fval$
      $\mathbf{gX} \leftarrow \mathbf{swarm[i]}$
    end if
  end for
  return $swarm$
end function

the step size α using a backtracking line search described in Algorithm 6.

Multiple optimizations increase the likelihood of converging to the global minimum. After running the optimizations from many points, we aggregate the results at each iteration to get the best one.

C. Automatic Differentiation

Our implementation of forward-mode AD relies on dual numbers to compute gradients accurately and efficiently. Dual numbers are written as

$$a + b\,\epsilon,$$

Algorithm 4 BFGS procedure with forward-mode AD

function SERIALBFGS($f, \text{range}, \text{iter}_{bfgs}, \mathbf{swarm[i]}, \Theta$)
  $\mathbf{x} \leftarrow \mathbf{swarm[i]}$, $k \leftarrow 0$ ▷ Set initial guess, loop counter
  $\mathbf{H} \leftarrow \mathbf{I}$ ▷ Identity matrix for Hessian
  while $k < \text{iter}_{bfgs}$ **do**
    $\nabla f(x) \leftarrow$ FORWARDAD(f, $\mathbf{x}$)
    if $\|\nabla f(x)\| < \Theta$ **then**
      $result.status = 1$;
      return $result$ ▷ Convergence criterion met
    end if
    $\mathbf{p} \leftarrow -\mathbf{H}\,\nabla f(\mathbf{x})$ ▷ Calculate search direction
    $\alpha \leftarrow$ LINESEARCH($f(\mathbf{x})$, $\mathbf{x}, \mathbf{p}, \mathbf{g}, \dim, \text{iter}_{ls}$)
    $\mathbf{x}_{\text{new}} \leftarrow \mathbf{x} + \alpha\,\mathbf{p}$ ▷ Update current point
    $\delta\mathbf{x} \leftarrow \mathbf{x}_{\text{new}} - \mathbf{x}$ ▷ Compute differences
    $\delta\mathbf{g} \leftarrow \nabla f(\mathbf{x}_{\text{new}}) - \nabla f(\mathbf{x})$
    Hessian update:

$$\mathbf{H}_{k+1} \leftarrow \left(\mathbf{I} - \frac{\delta\mathbf{x}\delta\mathbf{g}^{\mathbf{T}}}{\delta\mathbf{x}^{\mathbf{T}}\delta\mathbf{g}}\right)\mathbf{H_k}\left(\mathbf{I} - \frac{\delta\mathbf{g}\delta\mathbf{x}^{\mathbf{T}}}{\delta\mathbf{x}^{\mathbf{T}}\delta\mathbf{g}}\right) + \frac{\delta\mathbf{x}\delta\mathbf{x}^{\mathbf{T}}}{\delta\mathbf{x}^{\mathbf{T}}\delta\mathbf{g}}$$

    $\mathbf{x} \leftarrow \mathbf{x}_{\text{new}}, \quad k \leftarrow k + 1$
  end while
  $result.status = 0$
  return $result$
end function

where a and b are real numbers and ϵ is a symbol with the property

$$\epsilon^2 = 0, \epsilon \neq 0.$$

This property means that any term involving ϵ^2 vanishes. When using dual numbers for automatic differentiation, the coefficient b represents the partial derivative of a function. Then, we evaluate the objective function using our overloaded operators for dual numbers. Algorithm 5 describes the implementation of this approach by setting the value to $x[i]$ and the tangent or derivative part to 1.

Algorithm 5 Forward-Mode AD Procedure

function FORWARDAD(f, $x \in \mathbb{R}^{\text{dim}}$)
  Initialize gradient[dim]
  for $i \leftarrow 0$ to dim $- 1$ **do**
    $xDual[i] \leftarrow x[i] + 0\,\epsilon$ ▷ initializing dual number
  end for
  for $i \leftarrow 0$ to dim $- 1$ **do**
    Set $xDual[i].dual \leftarrow 1$ ▷ Seed derivative for variable i
    $result \leftarrow f(xDual)$
    gradient[i] $\leftarrow result.dual$
    Reset $xDual[i].dual \leftarrow 0$
  end for
  return $gradient$
end function

D. Line Search

The main goal of the line search in optimization algorithms is to select an optimal step size, α, that minimizes the objective function along a given search direction $\mathbf{p}$. The choice of this algorithm greatly influences the outcome of the optimization. Our algorithm implements a commonly used backtracking line search with the Armijo condition [Armijo(1966)], where it initially starts with a step size of 1. For the current point $x^{(i)}$, we aim to find the α such that the new point $\mathbf{x^{(i)}}_{\text{new}} = \mathbf{x^{(i)}} + \alpha^{(i)}\,\mathbf{p^{(i)}}$ yields the greatest reduction in function value $f(\mathbf{x^{(i)}})$.

The Armijo condition [Armijo(1966)] for backtracking is:

$$f(\mathbf{x} + \alpha\,\mathbf{p}) \leq f(\mathbf{x}) + c_1\,\alpha\,(\nabla f(\mathbf{x})^\top \mathbf{p}),$$

where c_1 is a small constant that we fixed to 0.3. The line search we used relies on this balance between large steps for faster progress and little steps to avoid overshooting. We found that twenty iterations is sufficient, and therefore we fixed that parameter. The following section gives an overview of the approach to GPU parallelization.

Algorithm 6 Backtracking line search using Armijo condition

```
1:  function LINESEARCH(f, x, p, g, iter_ls)
2:      c_1 ← 0.3, α ← 1.0
3:      ddir ← DOTPRODUCT(g, p)
4:      for i ← 0 to iter_ls do
5:          xTemp ← x + α p
6:          f_1 ← f(xTemp)
7:          if f_1 ≤ f_0 + c_1 α ddir then
8:              break
9:          end if
10:         α ← 0.5 × α
11:     end for
12:     return α
13: end function
```

IV. Approach to GPU parallelization

This section describes how we utilize parallelism to speed up each of the components of the sequential algorithm. In Algorithm 1, we parallelize the naturally independent phases of the sequential algorithm. The PSO initialization and each iteration can be done in parallel, and then the BFGS for each independent optimization. In both Algorithm 1, and Algorithm 7 the loop starting in Line 6 cannot be parallelized since each iteration depends on its previous iterations' velocities and best locations. Each thread randomly initializes the particles, runs the PSO main loop, and then applies BFGS until enough optimizations have converged. For this study, the parallelism we explored is on the first level. More levels of parallelism will be further studied in future work.

A. Parallel RNG

To initialize the starting points in Algorithm 10 we generated the random numbers on the GPU at runtime using the cuRAND library [Cook(2012)] by NVIDIA. We integrated an on-demand strategy where each thread generates its own starting point at runtime. For each dimension, a thread produces one double-precision number uniformly sampled from a user-given range $[lower, upper]$ using a unique seed based on its thread's global index plus the current dimension. This approach eliminates the need to store the entire size of $N \times dim$ array in memory.

Algorithm 7 ZEUS Parallel PSO-BFGS

```
1:  procedure ZEUS(f, N, range, iter_bfgs, iter_pso, iter_ls,
    required_c, Θ, w, c_1, c_2)
2:      Allocate swarm[N * dim], V[N * dim], pX[N * dim], pB[N], gX[dim];
3:      converged ← 0;                    ▷ number of converged runs
4:      gF ← +∞;
5:      PSOINITKERNEL(f, swarm, range, V, pX, pF, gX, gF, N)
6:      for i = 0 to iter_pso do
7:          PSOITERKERNEL(f, swarm, range, w, c_1, c_2, V, pX, pF, gX, gF, N)
8:      end for
9:      BFGSKERNEL(f, range, iter_bfgs, iter_pso, iter_ls, swarm, Θ, required_c, converged, stopflag)
10:     (best, idx) ← PARALLELREDUCTION({swarm[i]}_{i=1}^N)
11:     return best
12: end procedure
```

B. Parallel PSO

The way we handled parallelism can be broken down into two main phases: (1) the initialization of the particles and (2) the iterations of the swarm. Algorithm 8 initializes the swarm in parallel for each particle and determines the global best using atomic operations. Algorithm 9 is being used to update positions and velocities for each particle and find the global best at each iteration using atomic operations. Intuitively, it might seem that more iterations would yield superior accuracy. In Section V-E we show that this is not the case for all function types.

C. Parallel BFGS

Our approach executes a single optimization per thread, utilizing the extremely parallel nature of GPUs. Each thread gets a piece of the entire swarm array of size $N \times dim$. By starting from many points in parallel, our aim is to explore a large portion of the search space simultaneously.

In this implementation, we are synchronizing the threads so that if we hit the number of required converged optimizations, we let all threads know after to prevent wasting resources. At each iteration, we check to see if anyone has hit the $stopFlag$. Once the stop flag is hit, each thread currently working will see it at the next iteration. Then, we used CUDA's reduction kernel to find the global best from all of the results, as it provides an easily accessible kernel function [Cook(2012)].

We use the same line search and forward AD method as in the sequential algorithm. While the AD stage can in principle be parallelized because each partial derivative is independent, our

Algorithm 8 PSO Initialization Kernel

procedure PSOINITKERNEL($f, \mathbf{swarm}, \text{range}, \mathbf{V}, \mathbf{pX}, pF, \mathbf{gX}, gF, N$)
  for $i \leftarrow 0$ to $N-1$ **in parallel do**
    if $i \geq N$ **then**
      return
    end if
    $\Delta v \leftarrow (\text{upper} - \text{lower})$
    $\mathbf{r_x} \leftarrow \text{rand}(s, \text{lower}, \text{upper})$
    $\mathbf{r_v} \leftarrow \text{rand}(s, -\Delta v, \Delta v)$
    $\mathbf{swarm[i]} \leftarrow r_x$; $\mathbf{V[i]} \leftarrow \mathbf{r_v}$; $\mathbf{pX} \leftarrow \mathbf{r_x}$
    $fval \leftarrow f.\text{evaluate}(\mathbf{swarm[i]})$
    $pF[i] \leftarrow fval$
    $old \leftarrow \text{atomicMin}(gF, fval)$ ▷ atomic global-best
    if $fval < old$ **then**
      $\mathbf{gX} \leftarrow \mathbf{pX[i]}$
    end if
    $\text{states}[i] \leftarrow s$
  end for
end procedure

Algorithm 9 PSO Iteration Kernel

function PSOITERKERNEL($f, \mathbf{swarm}, \text{range}, w, c_1, c_2, \mathbf{V}, \mathbf{pX}, pF, \mathbf{gX}, gF, N$)
  for $i \leftarrow 0$ to $N-1$ **in parallel do**
    if $i \geq N$ **then**
      return
    end if
    $\mathbf{r_1} \leftarrow \text{rand}(s, 0, 1), \mathbf{r_2} \leftarrow \text{rand}(s, 0, 1)$
    $\mathbf{x} \leftarrow \mathbf{swarm[i]}, \mathbf{v} \leftarrow \mathbf{V[i]}$
    $\mathbf{p} \leftarrow \mathbf{pX[i]}, \ \mathbf{g} \leftarrow \mathbf{gX}$
    $\mathbf{v'} \leftarrow w\,\mathbf{v} + c_1\,\mathbf{r_1}\,(\mathbf{p} - \mathbf{x}) + c_2\,\mathbf{r_2}\,(\mathbf{g} - \mathbf{x})$
    $\mathbf{x'} \leftarrow \mathbf{x} + \mathbf{v'}$; $\mathbf{V[i]} \leftarrow \mathbf{v'}$ $\mathbf{swarm[i]} \leftarrow \mathbf{x'}$
    $fval \leftarrow f.\text{evaluate}(\mathbf{swarm[i]})$
    if $fval < pF[i]$ **then**
      $pF[i] \leftarrow fval$; $\mathbf{pX[i]} \leftarrow \mathbf{swarm[i]}$
    end if
    $old \leftarrow \text{atomicMin}(gF, fval)$
    if $fval < old$ **then**
      $\mathbf{gX} \leftarrow \mathbf{swarm[i]}$
    end if
    $\text{states}[i] \leftarrow s$
  end for
end function

Algorithm 10 BFGS Kernel with forward-mode AD

function BFGSKERNEL($f, \text{range}, \text{iter}_{bfgs}, \text{iter}_{pso}, \text{iter}_{ls}, \mathbf{swarm}, \Theta, required_c, \text{converged}, stopflag$)
  for $i \leftarrow 0$ to $N-1$ **in parallel do**
    $\mathbf{x} \leftarrow \mathbf{swarm}[i]$, $k \leftarrow 0$ ▷ Set initial guess, loop counter
    $\mathbf{H} \leftarrow \mathbf{I}$ ▷ Identity matrix for Hessian
    while $k <$ max_iter **do**
      if atomicAdd($stopFlag, 0$) $\neq 0$ **then** ▷ someone hit the stop flag
        break
      end if
      $\nabla f(x) \leftarrow$ FORWARDAD(f, $\mathbf{x}$)
      if $\|\nabla f(x)\| < \Theta$ **then**
        old $\leftarrow$ atomicAdd($converged, 1$)
        if old $=$ required$_c$ **then**
          atomicExch($stopFlag, 1$) ▷ first thread to reach the target sets stop
        end if
        break ▷ convergence criterion met
      end if
      $\mathbf{p} \leftarrow -\mathbf{H}\,\nabla f(\mathbf{x})$ ▷ Calculate search direction
      $\alpha \leftarrow$ LINESEARCH($f(\mathbf{x})$, $\mathbf{x}, \mathbf{p}, \mathbf{g}, \dim, \text{iter}_{ls}$)
      $\mathbf{x}_{\text{new}} \leftarrow \mathbf{x} + \alpha\,\mathbf{p}$ ▷ Update current point
      $\delta\mathbf{x} \leftarrow \mathbf{x}_{\text{new}} - \mathbf{x}$ ▷ Compute differences
      $\delta\mathbf{g} \leftarrow \nabla f(\mathbf{x}_{\text{new}}) - \nabla f(\mathbf{x})$
      Hessian update:

$$\mathbf{H}_{k+1} \leftarrow \left(\mathbf{I} - \frac{\delta\mathbf{x}\delta\mathbf{g}^{\mathbf{T}}}{\delta\mathbf{x}^{\mathbf{T}}\delta\mathbf{g}}\right)\mathbf{H_k}\left(\mathbf{I} - \frac{\delta\mathbf{g}\delta\mathbf{x}^{\mathbf{T}}}{\delta\mathbf{x}^{\mathbf{T}}\delta\mathbf{g}}\right) + \frac{\delta\mathbf{x}\delta\mathbf{x}^{\mathbf{T}}}{\delta\mathbf{x}^{\mathbf{T}}\delta\mathbf{g}}$$

      $\mathbf{x} \leftarrow \mathbf{x}_{\text{new}}$, $k \leftarrow k+1$
    end while
    return x
  end for
end function

performance measurements show that the Hessian update step dominates the BFGS kernel runtime as the problem dimension increases. In contrast, the AD component accounts for only a small fraction of the total runtime.

V. Experiments and Results

A. Experimental Setup

In this subsection we provide details about the experimental setup, including hardware and software used throughout the study. We used the same compute cluster to obtain both sequential and parallel results, where we used an Intel Xeon Gold 6148 CPU @ 2.40 GHz. On this same server we allocated an NVIDIA A100 device with 80GB of VRAM. This device has 6912 CUDA cores. To compile our code, we used CUDA 12.1.

We have utilized four commonly used benchmark functions described in the next subsection. For each function the error was calculated using the Euclidean distance that measures the distance between the estimated coordinates and the actual coordinates of the global minimum. When identifying a desirable error, we set the threshold to be 10^{-6}. The performance of our algorithm depends mainly on the characteristics of the functions being optimized and the number of dimensions. For this reason, we experimented with four widely used objective functions to test the performance of baseline methods and our approach.

B. Test Functions

The following test functions try to cover the spectrum of test functions. On convex surfaces like the Rosenbrock and Golstein-Price functions, our algorithm converges using a single optimization. However, for surfaces with many local minima for the Rastrigin and Ackley functions, many optimizations are needed to converge, especially in high-dimensional spaces.

1) Rosenbrock: The Rosenbrock function was proposed in 1960 and has been widely used to test the effectiveness of mathematical optimization algorithms [Rosenbrock(1960)]. It is a summation, where the number of dimensions can be increased dynamically, and is given by:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \left[(1 - x_i)^2 + 100 \cdot (x_{i+1} - x_i^2)^2\right]$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ and $\mathbf{x} \in \mathbb{R}^n$

2) Rastrigin: The Rastrigin test function was first proposed in 1974 [Rastrigin(1974)]. It is often used to test the efficiency of optimization algorithms in terms of convergence to the global minimum and execution time due to its highly multimodal nature. The number of local minima grows exponentially with the number of dimensions, which is shown in Figure 1.

The Rastrigin function in N dimensions is defined as:

$$f(\mathbf{x}) = A \cdot N + \sum_{i=1}^{N} \left[x_i^2 - A \cdot \cos(2\pi x_i)\right]$$

$$\text{where } A = 10, \quad \mathbf{x} \in \mathbb{R}^N$$

The function has a periodic pattern with local minima occurring at integer coordinates. The global minimum is at the origin. In the range, $[-5.12, 5.12]$ there are 11 integer values, which means in this space there are 11^2 or 121 local minima, with one being the global minimum.

3) Ackley: The Ackley function is a widely used function to test mathematical optimization algorithms. The function was proposed by David Ackley [Ackley(2012)]. It is defined for general d-dimensions:

$$f(x) = -20\exp\Big(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\Big) - \exp\Big(\frac{1}{d}\sum_{i=1}^{d}\cos(2\pi x_i)\Big) + e + 20$$

It has a similar landscape to the Rastrigin function with many local minima due to the periodic nature of the cosine function. In 2 dimensions, it has a single global minimum at (0.0, 0.0) where the function value is 0. However, the derivatives are undefined at (0.0, 0.0). For functions of this type we acknowledge the "failure" mode of our algorithm, where the program returns diverged status, where we reach the maximum number of iterations without converging based on the $\|\nabla f(x)\| < \Theta$ criterion. This behavior is illustrated in Figure 6. Future work include resolving this issue. This function violates the condition on continuity of derivatives. The algorithm never knows it has converged, never reaches a location where $\|\nabla f(x)\| < \Theta$.

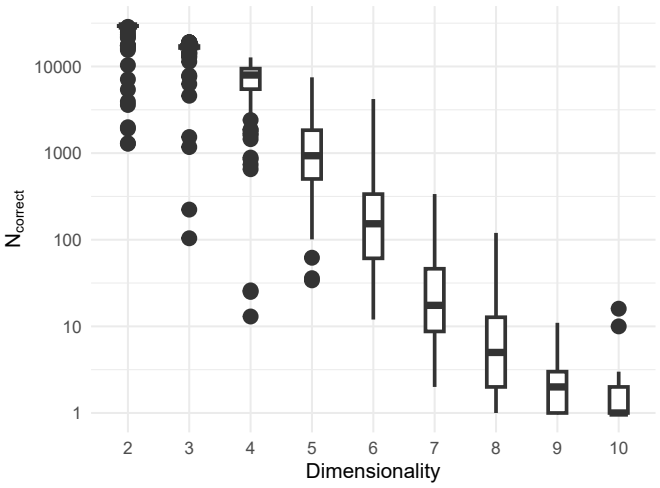

Fig. 1. Box and whisker plot showing performance degrades drastically for the Rastrigin function as the dimensionality of the problem increases when using the same number of particles. For each dimension, we plot 100 runs, where each run is a result of using 10^5 particles and 5 PSO iterations. We count the number of correct solutions across each run. N_{correct} corresponds to the count of each optimizations where the Euclidean error is less than 0.5.

4) Goldstein-Price: This function was first developed by Goldstein [Goldstein and Price(1971)]. It has a single minimum, but many starts require an infeasible number of steps to converge. It is defined as:

$$\begin{aligned} f(\mathbf{x}) = \big[1 + (x_1 + x_2 + 1)^2 (19 - 14x_1 + 3x_1^2 - 14x_2 \\ +6x_1x_2 + 3x_2^2)\big] \quad \times \\ \big[30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 \\ -36x_1x_2 + 27x_2^2)\big] \end{aligned}$$

Using these test functions, we have conducted experiments of two flavors. The first one describes the effectiveness of the algorithm, which is the ability to find the global minimum, the use of PSO, and BFGS. The second describes the GPU performance benefits. There is a need for multistart for several reasons.

C. Finding the right solution

In this experiment, we demonstrate the need for multistart approaches for functions with many local minima. If there are multiple local minima in the hyperspace like the Rastrigin function, as dimensionality grows we need more and more starting points to be confident in convergence to the global minimum. The 5-dimensional variant has 11^5 or $161,051$ local minima with a single global minimum. For the 10-dimensional Rastrigin function, the number of local minima in our searching space is 11^{10} or 26 billion, which means we would require to launch vastly more particles. A practitioner using the ZEUS algorithm would also need more than one convergences to have confidence we have the correct one. We describe future work how to handle this.

In Figure 1, we plot the distribution of a 100 runs using box and whiskers for each dimension, the number of particles

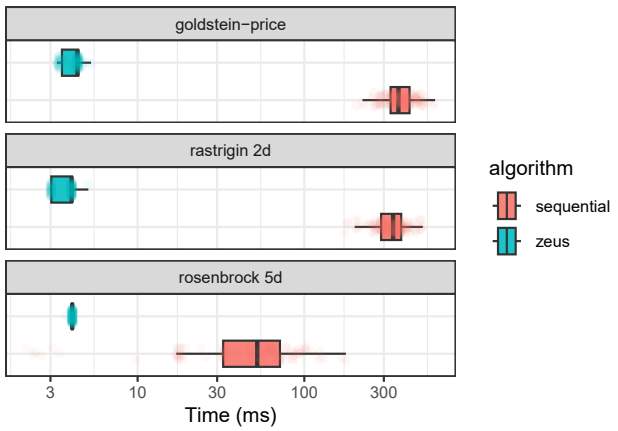

Fig. 2. Visual illustration of the speed advantage achieved by ZEUS for 2-dimensional and 5-dimensional objective functions. CPU runtimes were divided by the number of cores to approximate the ideal parallel execution. The distributions are based on 100 runs. Vertical jitter was applied to each point to make them more visible for ZEUS. The Ackley function was left out due to its misbehavior shown in Figure 6.

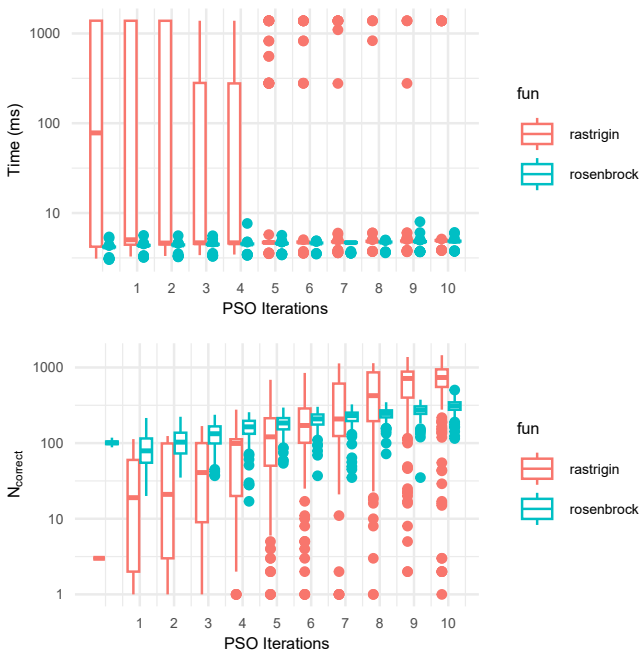

Fig. 3. Performance plots in terms of time (top) and number of correct solutions (bottom) across 100 runs compared with the PSO iterations for 5-dimensional Rastrigin (orange) and Rosenbrock (blue) functions. The Rastrigin function in this dimension has 11^5 or $161,051$ local minima. N_{correct} corresponding to the count of each optimization where the Euclidean error is less than 0.5.

that converge into the basin of the true global minimum. The plot shows results where each run went until the algorithm claims convergence 100 times in different dimensions of the Rastrigin function. The distribution of successful counts rapidly decreases with each added dimension, showing the exponential growth of local minima. For the first couple dimensions, the box is a line, as the distribution of the 100 runs are close together. With each dimension, a practitioner should trust the solution less and less. By ten dimensions, the number of correct solutions is effectively zero.

D. Parallel multistart for speed

By running many starts in parallel and stopping once a set number have converged, ZEUS cuts the time by orders of magnitude versus the fully sequential implementation. Running the sequential algorithm becomes infeasible for anything greater than a 2-dimensional Rastrigin function because it requires too many starts. For functions like the 2-dimensional Rosenbrock and Goldstein-Price function, BFGS will eventually converge from anywhere if we let it run long enough.

One of the advantages of the parallel algorithm is that it does not suffer from individual starting points that are far from the right solution. Whereas in the sequential variant, we must keep looping until we have enough converged optimizations. As a result, we can observe multiple orders of magnitude difference in the time it takes to converge for 100 optimizations using a small swarm of 1024. Figure 2 visually demonstrates this speedup between the sequential and parallel ZEUS. For fair comparison, the times for the sequential algorithm were divided by the number of cores of the CPU used for this experiment. The box and whisker plot shows a large difference for the 2-dimensional problems that only increases with the dimensionality of the problem. However, for the ZEUS algorithm, it is wasteful to use 1024 threads, as we are leaving many cores idle.

E. Improving the starting points by PSO

In this experiment, we show that it is useful to do a handful of PSO iterations as we increase the number of correct solutions and decrease the time it takes to converge to the required number of convergences. We show which functions benefit more from PSO iterations. Once we have enough converged particles where the norm of the gradient is sufficiently small enough, then BFGS synchronizes all other threads to stop early. Figure 3 demonstrates that PSO helps optimize functions with many local minima, and it is not too much of a waste of resources for problems with landscapes that have flat regions. In practical use cases, the user does not have information about the landscape of the function being optimized. The plot illustrates that for the Rastrigin function, the time it takes to gather enough convergences goes down with the number of PSO iterations (top), whereas the Rosenbrock function still benefits from more PSO computation in terms of the number of correct solutions (bottom). We can observe that we find more correct solutions as we increase the number of PSO iterations. For the Rastrigin function, we increased the number of correct solutions by multiple orders of magnitude.

F. Comparison with other libraries

The section compares the algorithm's performance against a Julia library in terms of error and time, and discusses how multi-start algorithms can utilize parallel execution on GPUs for faster convergence. In order to make the problem more complex, we have increased the number of dimensions to

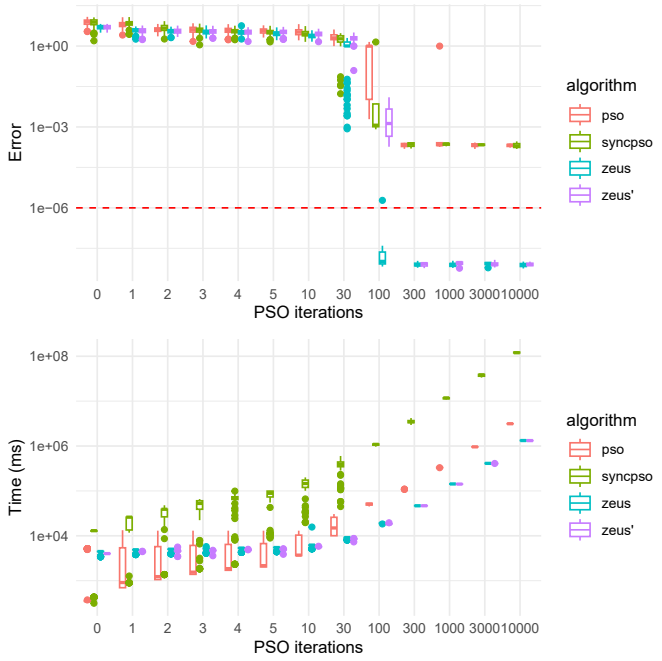

Fig. 4. Comparative performances for the 10-dimensional Rastrigin function with 11^{10} local minima. For a 10-dimensional Rastrigin to have 1 for the Euclidean error, means the point landed in a local minima, not the local minima.

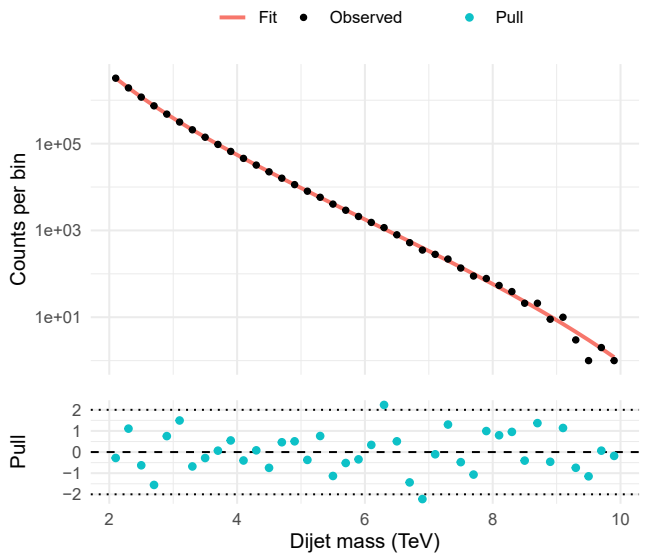

Fig. 5. Simulated dijet mass spectrum and fitted dijet mass spectra. The top panel shows the simulated event counts (black points) compared with the fitted prediction (red line). The bottom panel plots the pull distribution (blue points), defined as $\frac{N_{\text{obs}}-N_{\text{pred}}}{\sigma}$, where N_{obs} is the simulated count of events, N_{pred} is the model prediction, and σ is the statistical uncertainty per bin. The pulls fluctuate around zero and lie mostly within $\pm 2\sigma$, indicating agreement between simulation and prediction.

be 10 for this experiment. With this experiment we aim to gain insight into convergence behavior as we increase the number PSO iterations. We were able to use two different algorithms from their library. One of them is a parallel PSO with a synchronization point at each step, the other is a method that they note has race conditions. We do not consider using the asynchronous variant because it lacks the global update We can observe from Figure 4 that the algorithm has a degraded accuracy as we increase the number of steps compared to its other variant. This might be related to the race condition they mention, but further investigation should be done to confirm. After extensive communication with the authors, we were unable to get their Hybrid algorithm that uses BFGS to function. Since they have hard-coded the hyperparameters, we use the same ones to draw a fair comparison. We mark this algorithm as ZEUS'. The ZEUS algorithm uses hyperparameters derived from a paper [Deboucha et al.(2020)].

From the figure we can observe that ZEUS outperforms the library that has two of the same ingredients.

G. Real-world application

An important real-world application of numerical optimization is model fitting: given a numerical model with a set of parameters, to determine the best set of parameters to fit the data. The goal of such fitting is to constrain the model so that it can accurately predict future data. One way to evaluate the quality of the fit is to compare the observed data with the fitted model's prediction. In figure 5 shows the fitting of a simulated dijet mass spectrum and evaluation of the quality of the fit. The top panel compares the observed count of events to the prediction, showing that ZEUS finds fit parameters that yield an accurate prediction of the simulated data. The bottom panel shows the so-called *pull* distribution, which quantifies the deviations of each bin from the prediction relative to its statistical uncertainty. The pulls are centered around zero, with most values within $\pm 2\sigma$, indicating that the residuals are consistent with random statistical fluctuations and confirming the excellent quality of the fit.

VI. PROBLEM WITH CONVERGENCE

BFGS is controlled by the criterion of $\|\nabla f(x)\| < \Theta$. This condition works well for well-behaved functions with continuous first-order derivatives. For a function with discontinuous derivatives, the minimum may be located at a discontinuity. If the user sets the threshold to be too small, our program returns that we have not found the global minimum by the condition of the norm of the gradient. This behavior is illustrated in Figure 6, where the algorithm claimed convergence for optimizations landing in local minima where the condition $\|\nabla f(x)\| < \Theta$ was satisfied. However, points in or near the basin of the global minimum are stopped early because their norm was not below the threshold set by the user. Future work will handle functions with discontinuous derivatives.

VII. DISCUSSION

A. Performance Analysis

Our experiments demonstrated several clear patterns in how ZEUS behaves across different test functions. On the Rastrigin

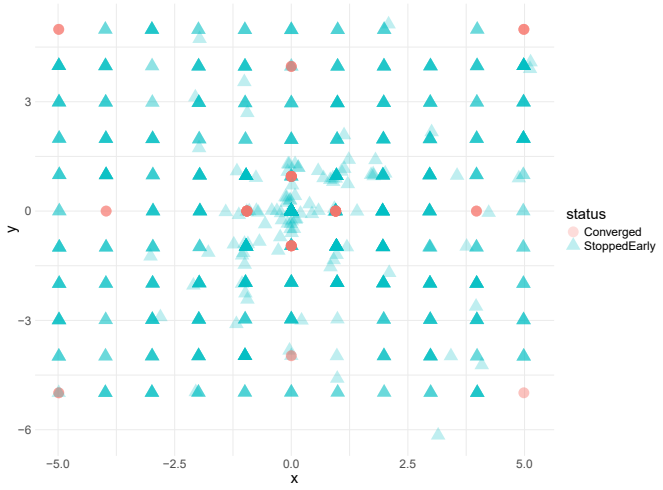

Fig. 6. Misbehavior of the algorithm claiming convergence for a 2-dimensional Ackley function for points where the norm of the gradient is sufficiently small enough and not declaring convergence for particles that are near the global minimum.

problem, we saw that as we increase the number of dimensions from two up to ten, the number of particles that actually find the global minimum falls close to zero. This happens because the number of local minima increases exponentially, which means the chance of landing in the correct basin rapidly vanishes.

By running many BFGS optimizations in parallel and stopping as soon as a set number of threads have converged, ZEUS achieves significant speedups over the fully sequential approach. In our experiment, the time dropped by one to two orders of magnitude on both two- and five-dimensional functions. The higher the dimension, the more benefit we gain by the parallel algorithm.

We also observed that our simple gradient-norm based convergence criterion can misfire on functions with discontinuous derivatives like the Ackley function. In such cases, particles may declare convergence too early when the norm of the gradient is small enough, but the point still may be far from the true minimum. Addressing this will require more sophisticated stopping criteria to handle functions with discontinuous derivatives in future versions of ZEUS.

B. Future Work

Future work will focus on exploiting the additional parallelism available in the single BFGS computation and reducing the computational complexity of the BFGS kernel itself. While parallelizing AD remains an option, our measurements show that the Hessian update dominates the runtime as the dimension grows. One promising approach is to explore L-BFGS [Liu and Nocedal(1989)], which reduces the complexity of the update step while maintaining quasi-Newton convergence behavior. This could significantly improve scalability, though potentially at some cost in solution accuracy due to the limited curvature information.

We will also explore means of improving the information given to the user about the degree of certainty in the reliability of the solution. By clustering candidate solutions found by the multistart algorithm, we can identify candidate regions for the local minima. If enough particles have been found to converge to the candidate region that has the lowest function value, and if no lower function value has been found, then we can have greater confidence that the region in question is the global minimum. We will consider using an iterative process that continues until a user-settable number of particles have converged to the same lowest region. We will explore two methods of clustering of the solutions based on their function value or their coordinates.

VIII. CONCLUSION

We have developed a GPU-accelerated algorithm, ZEUS, that has two main phases. The first phase selects random starting points which are then improved by using a PSO algorithm to move the particles to more promising regions. The second phase is a gradient-based BFGS algorithm that uses forward-mode AD to calculate the gradient at each iteration. No algorithm is universally best for every function, and the effectiveness of our algorithm depends on the specific function. When using a gradient-descent algorithm like BFGS, for problems in which the presence of multiple local minimal is suspected, the use of multiple starting points is required. Increasing the number of local minima increases the number of starting points needed to achieve confidence that the global minimum has been found. Random selection of starting points is typically used to provide a good probability for several of the BFGS searches to converge to the global minimum. Our results show that a few iterations of the PSO can improve the random starting points leading to faster convergence of the BFGS algorithm. For highly multimodal objective functions like the Rastrigin function, PSO increases the fraction of starts that land in the basin of starting points that lead to the global minimum, whereas for unimodal objectives like the Rosenbrock function, BFGS alone would be sufficient if we let it run long enough. Our open-source implementation makes these strategies available on GPUs, allowing practitioners to minimize non-convex problems more quickly and confidently than single-threaded solvers.

REFERENCES

[Ackley(2012)] David Ackley. 2012. *A connectionist machine for genetic hillclimbing*. Vol. 28. Springer science & business media, New York, NY.

[Armijo(1966)] Larry Armijo. 1966. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of mathematics* 16, 1 (1966), 1–3.

[Barkalov and Gergel(2016)] Konstantin Barkalov and Victor Gergel. 2016. Parallel global optimization on GPU. *Journal of Global Optimization* 66 (2016), 3–20.

[Baydin et al.(2018)] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of machine learning research* 18, 153 (2018), 1–43.

[Boender et al.(1982)] C Guus E Boender, AHG Rinnooy Kan, GT Timmer, and Leen Stougie. 1982. A stochastic method for global optimization. *Mathematical programming* 22 (1982), 125–140.

[Bradbury et al.(2018)] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. Google. http://github.com/jax-ml/jax

[Broyden(1970)] Charles George Broyden. 1970. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics* 6, 1 (1970), 76–90.

[Bücker(2006)] Martin Bücker. 2006. *Automatic differentiation: applications, theory, and implementations*. Springer, Berlin, Heidelberg.

[Carpenter et al.(2015)] Bob Carpenter, Matthew D. Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. 2015. The Stan Math Library: Reverse-Mode Automatic Differentiation in C++. arXiv:1509.07164 [cs.MS] https://arxiv.org/abs/1509.07164

[Clerc and Kennedy(2002)] Maurice Clerc and James Kennedy. 2002. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE transactions on Evolutionary Computation* 6, 1 (2002), 58–73.

[Cook(2012)] Shane Cook. 2012. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, Burlington, Massachusetts.

[Csendes(1988)] Tibor Csendes. 1988. Nonlinear parameter estimation by global optimization-efficiency and reliability. *Acta Cybernetica* 8, 4 (1988), 361–370.

[Deboucha et al.(2020)] Houssam Deboucha, Saad Mekhilef, Sofia Belaid, and Amar Guichi. 2020. Modified deterministic Jaya (DM-Jaya)-based MPPT algorithm under partially shaded conditions for PV system. *IET Power Electronics* 13, 19 (2020), 4625–4632.

[Della Santa(2024)] Francesco Della Santa. 2024. Automatic Differentiation-Based Multi-Start for Gradient-Based Optimization Methods. *Mathematics* 12, 8 (2024), 1201.

[Dixit et al.(2024)] Vaibhav Kumar Dixit, Julian Samaroo, Avik Pal, Alan Edelman, Christopher Vincent Rackauckas, et al. 2024. Efficient GPU-Accelerated Global Optimization for Inverse Problems. In *ICLR 2024 Workshop on AI4DifferentialEquations In Science*. ICLR, Vienna, Austria.

[Ferreiro et al.(2019)] Ana M Ferreiro, José Antonio García-Rodríguez, Carlos Vázquez, E Costa e Silva, and Aldina Correia. 2019. Parallel two-phase methods for global optimization on GPU. *Mathematics and Computers in Simulation* 156 (2019), 67–90.

[Fletcher(1970)] Roger Fletcher. 1970. A new approach to variable metric algorithms. *The computer journal* 13, 3 (1970), 317–322.

[Goldfarb(1970)] Donald Goldfarb. 1970. A family of variable-metric methods derived by variational means. *Mathematics of computation* 24, 109 (1970), 23–26.

[Goldstein and Price(1971)] Allen A Goldstein and JF Price. 1971. On descent from local minima. *Mathematics of computation* 25, 115 (1971), 569–574.

[Grabner et al.(2008)] Markus Grabner, Thomas Pock, Tobias Gross, and Bernhard Kainz. 2008. Automatic Differentiation for GPU-Accelerated 2D/3D Registration. In *Advances in Automatic Differentiation*, Christian H. Bischof, H. Martin Bücker, Paul Hovland, Uwe Naumann, and Jean Utke (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 259–269.

[Hueckelheim et al.(2023)] Jan Hueckelheim, Harshitha Menon, William S Moses, Bruce Christianson, Paul Hovland, and Laurent Hascoet. 2023. A Short Review of Automatic Differentiation Pitfalls in Scientific Computing. In *ICML 2023 Workshop on Differentiable Almost Everything: Differentiable Relaxations, Algorithms, Operators, and Simulators*. ICML, 1269 Law Street, San Diego CA 92109.

[Jain et al.(2022)] Meetu Jain, Vibha Saihjpal, Narinder Singh, and Satya Bir Singh. 2022. An overview of variants and advancements of PSO algorithm. *Applied Sciences* 12, 17 (2022), 8392.

[Kennedy and Eberhart(1995)] James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, Vol. 4. ieee, IEEE, Perth, WA, Australia, 1942–1948.

[Ketkar(2017)] Nikhil Ketkar. 2017. *Stochastic Gradient Descent*. Apress, Berkeley, CA, 113–132. https://doi.org/10.1007/978-1-4842-2766-4_8

[King(2009)] Davis E King. 2009. Dlib-ml: A machine learning toolkit. *The Journal of Machine Learning Research* 10 (2009), 1755–1758.

[Li et al.(2014)] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. 2014. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, New York New York USA, 661–670.

[Li et al.(2011)] Shutao Li, Mingkui Tan, Ivor W Tsang, and James Tin-Yau Kwok. 2011. A hybrid PSO-BFGS strategy for global optimization of multimodal functions. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 41, 4 (2011), 1003–1014.

[Liu and Nocedal(1989)] Dong C Liu and Jorge Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical programming* 45, 1 (1989), 503–528.

[Margossian(2019)] Charles C Margossian. 2019. A review of automatic differentiation and its efficient implementation. *Wiley interdisciplinary reviews: data mining and knowledge discovery* 9, 4 (2019), e1305.

[Moses et al.(2021)] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-mode automatic differentiation and optimization of GPU kernels via enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. https://doi.org/10.1145/3458817.3476165

[Nezhad et al.(2013)] Ali Mohammad Nezhad, Roohollah Aliakbari Shandiz, and Abdolhamid Eshraghniaye Jahromi. 2013. A particle swarm–BFGS algorithm for nonlinear programming problems. *Computers & operations research* 40, 4 (2013), 963–972.

[Noel(2012)] Mathew M Noel. 2012. A new gradient based particle swarm optimization algorithm for accurate computation of global minimum. *Applied Soft Computing* 12, 1 (2012), 353–359.

[Pu and Yu(1990)] Dingguo Pu and Wenci Yu. 1990. On the convergence property of the DFP algorithm. *Annals of Operations Research* 24, 1 (1990), 175–184.

[Rastrigin(1974)] Leonard Andreevič Rastrigin. 1974. *Systems of extremal control*. Theoretical Foundations of Engineering Cybernetics, Vol. 3. Nauka, Moscow.

[Reddi et al.(2016)] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex Smola. 2016. Stochastic variance reduction for nonconvex optimization. In *International conference on machine learning*. PMLR, PMLR, New York, New York, USA, 314–323.

[Rosenbrock(1960)] HoHo Rosenbrock. 1960. An automatic method for finding the greatest or least value of a function. *The computer journal* 3, 3 (1960), 175–184.

[Shanno(1970)] David F Shanno. 1970. Conditioning of quasi-Newton methods for function minimization. *Mathematics of computation* 24, 111 (1970), 647–656.

[Shi and Eberhart(1998)] Yuhui Shi and Russell Eberhart. 1998. A modified particle swarm optimizer. In *1998 IEEE international conference on evolutionary computation proceedings. IEEE world congress on computational intelligence (Cat. No. 98TH8360)*. Ieee, IEEE, Anchorage, AK, USA, 69–73.

[Singh et al.(2024)] Nitesh Kumar Singh, Ion Necoara, and Vyacheslav Kungurtsev. 2024. Mini-batch stochastic subgradient for functional constrained optimization. *Optimization* 73, 7 (2024), 2159–2185.

[Thobirin and Yanto(2015)] Aris Thobirin and Iwan Tri Riyadi Yanto. 2015. Automatic differentiation based for particle swarm optimization Steepest descent direction. *International Journal of Advances in Intelligent Informatics* 1, 2 (2015), 90–97.

[Wu et al.(2014)] Guohua Wu, Dishan Qiu, Ying Yu, Witold Pedrycz, Manhao Ma, and Haifeng Li. 2014. Superior solution guided particle swarm optimization combined with local search techniques. *Expert Systems with Applications* 41, 16 (2014), 7536–7548.

[Zhang et al.(2016)] PG Zhang, CL Yang, ZH Xu, ZL Cao, QQ Mu, and L Xuan. 2016. Hybrid particle swarm global optimization algorithm for phase diversity phase retrieval. *Optics Express* 24, 22 (2016), 25704–25717.

[Zinkevich et al.(2010)] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. 2010. Parallelized stochastic gradient descent. *Advances in neural information processing systems* 23 (2010).

[Zubair et al.(2023)] Mohammad Zubair, Desh Ranjan, Aaron Walden, Gabriel Nastac, Eric Nielsen, Boris Diskin, Marc Paterno, Samuel Jung, and Joshua Hoke Davis. 2023. Efficient GPU Implementation of Automatic Differentiation for Computational Fluid Dynamics. In *2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, IEEE, Goa, India, 377–386.