An Augmented Lagrangian Interior-Point Approach for Large-Scale NLP Problems on Graphics Processing Units

Yankai Cao, Arpan Seth, Carl D. Laird*

School of Chemical Engineering, Purdue University, 480 Stadium Mall Drive, West Lafayette, IN 47907, USA

Abstract

The demand for fast solution of nonlinear optimization problems, coupled with the emergence of new concurrent computing architectures, drives the need for parallel algorithms to solve challenging nonlinear programming (NLP) problems. In this paper, we propose an augmented Lagrangian interior-point approach for general NLP problems that solves in parallel on a Graphics processing unit (GPU). The algorithm is iterative at three levels. The first level replaces the original problem by a sequence of bound-constrained optimization problems using an augmented Lagrangian method. Each of these bound-constrained problems is solved using a nonlinear interior-point method. Inside the interior-point method, the barrier sub-problems are solved using a variation of Newton's method, where the linear system is solved using a preconditioned conjugate gradient (PCG) method, which is implemented efficiently on a GPU in parallel. This algorithm shows an order of magnitude speedup on several test problems from the COPS test set.

1. Introduction

Fast solution of nonlinear programming (NLP) problems is important in a number of different application areas, including many online or real-time applications. However, over the past decade, we have seen a fundamental change in computing hardware, and previously observed exponential increases in CPU clock rate have stagnated. While clock rate is not the only determining factor in CPU performance, it is clear that CPU manufacturers have shifted their focus towards multi-core and other parallel computing architectures even for everyday computing needs. The need for increasingly powerful NLP solvers, coupled with the introduction of low-cost parallel computing architectures, heightens the need for the development of NLP algorithms that can utilize these emerging parallel architectures.

According to Flynn's taxonomy, there are two typical parallel architectures: multiple-instruction-multiple-data (MIMD) architectures and single-instruction-multiple-data (SIMD) architectures. MIMD architectures can simultaneously execute different instructions in parallel. Two relatively popular classes of MIMD architectures are distributed computing clusters (e.g. beowulf cluster) and multi-core (or shared memory) machines. A distributed computer

*Corresponding author.

Email address: lairdc@purdue.edu (Carl D. Laird)

is typically composed of more than 100 processors connected by a network. For traditional desktop computing needs, the price of distributed computers are generally prohibitive. Furthermore, the communication overhead between processors can lead to efficiency bottlenecks. On the other hand, a multi-core or shared memory architecture provides multiple processing units that share common memory within a single machine. This reduces communication costs, but the possible gains in performance are limited by the number of cores, which is typically small, and potential bottlenecks with multiple processes accessing shared memory.

With Single-Instruction-Multiple-Data (SIMD) architectures, each thread can operate on different data, but must execute the same fundamental instruction. Graphics processing units (GPUs), one of the most widely used SIMD architectures available, are emerging as massively parallel systems that offer a large degree of parallelism at relatively low cost. For example, the NVIDIA GPU Tesla K20X, which sold for approximately three thousand US dollars provides 2688 cores capable of 3.95 teraflops for single-precision operation and 1.31 teraflops for double-precision operation. GPUs like the NVIDIA Tesla K20X are not pure SIMD architectures, but rather a hybrid architecture that is composed of a number of independent Streaming Multiprocessors (SMs), each containing several CUDA (Compute Unified Device Architecture) cores. Nevertheless, efficient performance requires that we make use of the parallelism available in the SIMD components and it is reasonable to consider them as such. GPUs offer a much higher level of parallelism than desktop multi-core architectures, are much cheaper than distributed computers, and can be programmed using the rapidly maturing GPU APIs integrated with commonly used languages like C and C++. These advances in GPU technology make it a viable and highly accessible architecture upon which to build numerical algorithms. Over the past several years, GPUs have been used to solve problems in various fields, such as air pollution(Jr. et al., 2010), elasticity simulation (Dick et al., 2011), computational fluid dynamics (Corrigan et al., 2011), neurosurgical simulation (Roman et al., 2010), risk assessment (Zhang et al., 2011), partial differential equations (Elble et al., 2010), and bioinformatics (Vouzis and Sahinidis, 2011). However, so far there has been little research done on developing general NLP algorithms for GPUs. In this paper we seek to develop an algorithm for NLP problems that can make use of the massive, affordable parallel computing capacity of GPUs.

Restricting ourselves to the class of large-scale continuous nonlinear optimization problems, interior-point methods are among the most successful general purpose algorithms. The dominant computational expense in these NLP algorithms is the solution of a large, sparse linear system to generate the step direction at each iteration. A scalable parallel algorithm requires efficient parallel solution of this linear system in addition to parallel evaluation of the NLP functions and gradients. There is the possibility for parallel solution of these linear systems using either direct or iterative methods. Amestoy et al. (2000) presents a parallel distributed memory multifrontal approach to solve sparse linear equations. A speedup of more than 7 is achieved on some test problems with this algorithm. However, the speedup is shown to stagnate with more than 32 processors for most of the test problems. Schenk and Gärtner (2004) shows a parallel sparse unsymmetric LU factorization method integrated into the PARDISO solver for use on shared memory multiprocessor architectures, achieving a speedup of more than 7 on a 8-core machine. Hogg and Scott (2010) develops a symmetric indefinite sparse direct solver within the HSL library for use on multicore machines with OpenMP, achieving a speedup of more than 6 on a 8-core machine. The speedup possible

with these approaches is promising, however, the available parallelism is too small for the GPU. Recently, breakthroughs have been made by several researchers (Krawezik and Poole, 2010; Lucas et al., 2012), who applied a multifrontal factorization method on a GPU. The multifrontal method factorizes a sparse matrix by factorizing a tree of dense systems, each of which can be implemented efficiently on a GPU (Galoppo et al., 2005; Agullo et al., 2009; Tomov et al., 2010; Cao et al., 2013). A speedup factor of 2-3 is reported for double precision on a matrix with more than half a million rows/columns. Yeralan et al. (2013) pushes the research further by factorizing those frontal matrices in parallel and achieved up to 10 times speedup on their test problems. However the speedup is not always consistent. For some problems, this algorithm provides limited parallelism, therefore the implementation on a GPU can perform even worse than that on a CPU. In contrast to direct factorization techniques, iterative methods, which require performing simple matrix-vector operations on consistently structured data sets, are highly appropriate for the GPU architecture. The PETSc Library has GPU support for many Krylov Subspace methods with Jacobi, AMG (Algebraic Multigrid), and AINV (Approximate Inverse) preconditioners (Kumbhar, 2011). Among iterative methods, the Preconditioned Conjugate Gradient (PCG) method is known to have excellent performance, and several researchers have demonstrated up to 10 times speedup using PCG approaches with different preconditioners on GPUs (Li and Saad, 2013; Helfenstein and Koko, 2011; Buatois et al., 2009).

However, the desire to use PCG to solve the linear system imposes limitations on the NLP algorithm we can use. PCG requires the matrix in the linear system to be positive definite (P.D.) (Golub and Van Loan, 2012). But Karush-Kuhn-Tucker (KKT) systems arising from many NLP algorithms are not P.D. For example, in IPOPT (Wächter and Biegler, 2006), the KKT system arising from the interior-point method is a saddle point system and is indefinite even for convex problems.

$$\begin{bmatrix} W_k + \Sigma_k & A_k \\ A_k^T & -D \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -\varphi(x_k) + A_k \lambda_k \\ c(x_k) \end{bmatrix}$$

Hence, PCG cannot be directly applied to this saddle point system. For a saddle point system with $D=0$, Bergamaschi et al. (2004); Lukšan and Vlček (1998); Perugia and Simoncini (2000); Gould et al. (2001) employ a constraint preconditioner that allows the use of the PCG method. Dollar et al. (2006); Dollar (2007) extend the constraint preconditioner for general D . Forsgren et al. (2007) shows that the saddle point system with positive diagonal D can be transformed into the doubly augmented system or condensed system. Provided the original saddle point system has the correct inertia, the augmented system and condensed system can be proven to be positive definite, and the PCG method in conjunction with the constraint preconditioner shows promise. If the PCG method detects negative curvature, it implies the matrix is not positive definite and the inertia condition is not satisfied for the original system. However, a major problem with applying this technique is that the constraint preconditioner requires a sparse matrix factorization and backsolves, for which no currently efficient implementations are available for the general case.

The augmented Lagrangian method moves the equality constraints to the objective function and solve the NLP as a sequence of bound-constrained sub-problems. This method has the benefit that the KKT system is positive definite when the problem is convex. Lancelot (Conn et al., 1988), first released in 1992, is a well-known example of an augmented

Lagrangian code for NLP problems. Lancelot uses the gradient projection method to solve the bound-constrained problems. This method first finds the Cauchy point, which is the first local minimizer of the approximation of the objective function along the steepest descent direction (Conn et al., 1988), or a point satisfying sufficient decrease condition (Lin and Moré, 1999). If the bounds are reached before the first minimizer is found, the search direction is bent at the corresponding bounds. Then the gradient projection method fixes the components of the Cauchy point that are at their bounds and performs the subspace minimization with the PCG method. When the bounds are violated in a particular PCG iteration, the PCG method is terminated (Conn et al., 1988). With the gradient projection method, the PCG iterations are easily stopped because of the violation of the bounds. Since changes in the active set are performed on the host, this can result in many transitions back and forth from the host to the GPU, causing deterioration of parallel performance. Therefore, gradient projection method is not directly appropriate for parallel implementation on a GPU.

In our work, we use an interior-point method to solve the bound-constrained problems by replacing each with a series of unconstrained barrier sub-problems. We can use a variation of Newton’s method to solve the unconstrained sub-problem. The primary advantage of this approach is that we can use PCG to compute the Newton’s step in parallel on a GPU. For convex problems, the KKT matrix arising from the unconstrained sub-problem is guaranteed to be P.D. Furthermore, even for non-convex problems, when the variables are near the optimal solution the matrix is also P.D. However, when the variables are far away from the optimal solution, the matrix is not guaranteed to be P.D. Therefore, if the PCG approach detects negative curvature, a diagonal modification to the matrix can be made to ensure that the matrix is positive definite and the step direction is a descent direction.

The paper is organized as follows. A description of the NLP problem formulation and the overall solution is given in Section 2. Section 3 describes the parallel implementation details of the proposed algorithm, including a discussion of pros and cons for using different matrix storage formats. Section 4 presents the numerical performance of the algorithm on some problems selected from the COPS (Dolan et al., 2004) test set including a comparison with the state of the art solver IPOPT. Section 5 summarizes the paper.

2. Algorithm

2.1. Problem Formulation

The general nonlinear programming problem we consider is of the form

$$\min_{x \in \mathbb{R}^n} f(x) \tag{1a}$$

$$\text{s.t. } c(x) = 0 \tag{1b}$$

$$x_L \leq x \leq x_U, \tag{1c}$$

where the objective function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, and equality constraints $c : \mathbb{R}^n \rightarrow \mathbb{R}^m$ are twice continuously differentiable. The vectors x_L and x_U are the lower and upper bounds. For this discussion, at any local minimum x^* , we assume that the linear independence constraint qualification (LICQ) and second-order sufficient conditions (SOS) hold as given in Theorem 17.5 of Nocedal and Wright (2006). Problems with general inequality constraints can be

transformed to this form with the introduction of slack variables. The algorithm discussion in sections 2.2 - 2.5 uses a simplified problem notation of the form

$$\min_{x \in \mathbb{R}^n} f(x) \quad (2a)$$

$$\text{s.t. } c(x) = 0 \quad (2b)$$

$$x \geq 0. \quad (2c)$$

In section 2.6, we discuss the modifications necessary to handle the general form (1).

2.2. Bound-constrained Augmented Lagrangian Method

The augmented Lagrangian is formed by adding a quadratic penalty term to the Lagrangian function. Neglecting the inequalities, the augmented Lagrangian for eqs. (2a) and (2b) is

$$\mathcal{L}_A(x, \lambda; \mu) = f(x) - \lambda^T c(x) + \frac{\mu}{2} c(x)^T c(x), \quad (3)$$

where μ is the penalty parameter and λ is the estimate of the Lagrange multipliers. The augmented Lagrangian approach then computes the solution for a sequence of bound-constrained sub-problems

$$\min_{x \in \mathbb{R}^n} \mathcal{L}_A(x, \lambda; \mu) \quad (4a)$$

$$\text{s.t. } x \geq 0. \quad (4b)$$

This sub-problem is solved approximately for fixed value of λ and μ . If the violation of the equality constraints has decreased sufficiently, the estimate of the Lagrange multipliers λ is updated, and the sub-problem tolerance is made tighter. Otherwise, the penalty parameter μ is increased to improve feasibility.

It can be proven (Nocedal and Wright, 2006) that under reasonable assumptions, a good estimate of the optimal solution x^* can be obtained after solving a sequence of sub-problems when μ is large enough or when λ is a good estimate of the Lagrange multipliers. Also, if μ is sufficiently large, then each λ update will lead to a more accurate estimate of the Lagrange multipliers. It can also be proven that the Hessian of the augmented Lagrangian function is positive definite when x and λ are sufficiently close to the optimal solution.

The augmented Lagrangian framework forms the outer-loop of the algorithm. At each iteration, we require the solution of the bound-constrained sub-problem.

2.3. Interior-Point Method for the Bound-Constrained Sub-Problem

There are several techniques one could use to solve the bound-constrained sub-problem including an active set method, a gradient projection method, or an interior-point method. The use of an interior-point method produces a linear system that can be solved efficiently on a GPU. Hence, we replace the bound-constrained sub-problem by a series of unconstrained barrier sub-problems

$$\min_{x \in \mathbb{R}^n} \varphi(x) = \mathcal{L}_A - \mu_{in} \sum_{i=1}^n \ln(x^{(i)}), \quad (5)$$

where $x^{(i)}$ denotes the i th component of the vector x , and $\mu_{in} > 0$ is the barrier parameter. The first-order optimality conditions of the unconstrained sub-problem are

$$\nabla\varphi(x) = \nabla\mathcal{L}_A(x) - \mu_{in}X^{-1}e = 0, \quad (6)$$

where $X = \text{diag}(x)$, and e is a vector with all elements equal to 1. The primal-dual reformulation can be written by introducing $z = \mu_{in}X^{-1}e$,

$$\nabla\mathcal{L}_A(x) - z = 0 \quad (7a)$$

$$Xz - \mu_{in}e = 0. \quad (7b)$$

When μ_{in} is 0, the above equations together with $x \geq 0, z \geq 0$ are the KKT conditions for the bound-constrained sub-problem (4). We can use a variation of Newton's method to solve the primal-dual system of equations (7). At each iteration k of the Newton's method, the step direction is calculated by solving the linear system

$$\begin{bmatrix} W_k & -I \\ Z_k & X_k \end{bmatrix} \begin{bmatrix} d_k^x \\ d_k^z \end{bmatrix} = - \begin{bmatrix} \nabla\mathcal{L}_A - z_k \\ X_k z_k - \mu_{in}e \end{bmatrix}. \quad (8)$$

Here W_k denotes the Hessian of the augmented Lagrangian function

$$W_k := \nabla^2\mathcal{L}_A = \nabla^2 f(x_k) - \sum (\lambda_i - \mu c_i(x_k)) \nabla^2 c_i(x_k) + \mu A_k A_k^T, \quad (9)$$

with $A_k := \nabla c(x)$, while d_k^x and d_k^z are the search directions in x_k and z_k respectively. The first two terms of equation (9) can be expressed as $\nabla^2\mathcal{L}(x_k, \lambda - \mu c(x_k))$ while $\mathcal{L}$ is the Lagrangian function. A smaller and symmetric system can be obtained from equation (8) by multiplying the last block row by X_k^{-1} and adding it to the first block row

$$[W_k + \Sigma_k] d_k^x = -\nabla\varphi(x_k), \quad (10)$$

where $\Sigma_k = X_k^{-1}Z_k$. After solving equation (10) for d_k^x , the step in the multipliers d_k^z can be obtained using

$$d_k^z = \mu_{in}X_k^{-1}e - z_k - \Sigma_k d_k^x. \quad (11)$$

After the step directions are determined, the maximum step size $\alpha_k^{x,max}$ for primal variables and α_k^z for dual variables can be calculated based on the fraction-to-the-boundary rule to ensure $x > 0$ and $z > 0$. With $\alpha_k^{x,max}$ as the initial guess, a line search is performed to get the step size α_k^x for primal variables. Finally, the values of primal and dual variable for the next interior-point iteration are calculated by

$$x_{k+1} = x_k + \alpha_k^x d_k^x \quad (12a)$$

$$z_{k+1} = z_k + \alpha_k^z d_k^z. \quad (12b)$$

2.4. Using PCG to Solve the Linear KKT System

The linear system (10) that needs to be solved at each interior-point iteration is

$$H_k d_k^x = -\nabla \varphi(x_k), \quad (13)$$

where

$$H_k := W_k + \Sigma_k. \quad (14)$$

Solving this linear system is the dominant cost of the algorithm, and we want to use a PCG method since it can be parallelized on the GPU. Applying the PCG approach for solution of equation (13) requires H_k to be P.D. If the original problem (1) is convex, H_k is guaranteed to be P.D. and PCG can be applied to the system.

For non-convex problems, if H_k is P.D., then PCG can be applied directly. If H_k is not P.D. (which can be detected in the PCG steps), then the PCG approach is aborted, a diagonal modifier $\delta_w I$ is added to the H_k and the linear system is solved again. If we continue to detect negative curvature the value of δ_w is increased according to the rule described in Wächter and Biegler (2006).

Forming H_k explicitly using sparse matrix-matrix multiplication can be expensive. At each iteration, PCG only requires a series of matrix-vector products with H_k . Hence, in our implementation, H_k is not formed explicitly. Instead, the matrix vector products are performed across the right hand side expression in equation (15).

$$H_k := \nabla^2 \mathcal{L}(x_k, \lambda - \mu c(x_k)) + \mu A_k A_k^T + \Sigma_k + \delta_w I, \quad (15)$$

Therefore, each PCG step involves three sparse matrix-vector multiplications - $\nabla^2 \mathcal{L}$ with a vector, $[A_k]^T$ with a vector, and the multiplication of the resulting vector with A_k . This implicit implementation saves significant computational expense.

It is possible that the PCG method converges (based on its tolerance) even if the matrix H_k is not P.D. However, the starting point for d_k^x is set to zero, and the solution from the PCG approach is still guaranteed to be a descent direction for the barrier sub-problem (Dembo and Steihaug, 1983).

The PCG method can be accelerated with a suitable preconditioner. However it can be very challenging to efficiently implement many known preconditioners on a GPU since preconditioner factorization and backsolves are typically inefficient on the GPU (Li and Saad, 2013) (Naumov, 2011). Furthermore, in our algorithm the matrix H_k is never explicitly formed, limiting the choice for preconditioner. Therefore, a simple diagonal preconditioner is used.

2.5. Algorithm Summary

This section summarizes the overall algorithm for solving problem (2). The algorithm is iterative at three levels. The first level replaces the original problem by a sequence of bound-constrained optimization problems using an augmented Lagrangian method. Each of these bound-constrained problems is solved using a nonlinear interior-point method. Inside the interior-point method, the barrier sub-problems are solved using a variation of Newton's method, and the linear system is solved iteratively with a preconditioned conjugate gradient

method. In Algorithm 1 presented below, we provide the augmented Lagrangian method used in this paper. Following that, we present Algorithm 2 that describes the interior-point method used to solve the bound-constrained sub-problems in Algorithm 1.

Algorithm 1 : Augmented Lagrangian Method

1. Initialize

Initialize the iteration index $j \leftarrow 0$.

Set initial points (x_0, λ_0) with $x_0 > 0$; overall convergence tolerance for the equality constraints and sub-problems η_* and ω_* ; initial penalty parameter for j th iteration $\mu_j > 0$; initial tolerance for j th sub-problem ω_j and η_j .

2. Solve bound-constrained sub-problems

Use Algorithm 2 to find a minimizer (x_j^*, z_j^*) for (4) such that optimality error of the sub-problems j satisfies $E_0(x_j^*, z_j^*) \leq \omega_j$ as computed in step **2** of Algorithm 2.

3. Update penalty parameter, Lagrangian multiplier and tolerance

if $\|c(x_j)\| \leq \eta_j$ **then**

if $\|c(x_j)\| \leq \eta_*$ and $E_0(x_j^*, z_j^*) \leq \omega_*$ **then**

Stop with solution $(x^*, z^*, \lambda^*) \leftarrow (x_j^*, z_j^*, \lambda_j)$.

end if

Update multipliers λ_j and tighten tolerances η_j and ω_j .

else

Increase penalty parameter μ_j and update tolerance η_j and ω_j .

end if

Update $j \leftarrow j + 1$.

Return to step **2**.

The details about μ_j , ω_j , η_j initialization and how they are updated in step **3** are described in Conn et al. (1988). Now we provide the details of the interior-point algorithm to solve the j th sub-problem in step **2**.

Algorithm 2 : Interior-point Method

- 1. Initialize**
Initialize the iteration index $k \leftarrow 0$ and optimality tolerance ω_j from Algorithm 1.
Set starting point (x_0, z_0) with $z_0 > 0$; initial barrier parameter $\mu_{in} > 0$; tolerance constants $\kappa_\epsilon > 0$.
 - 2. Check convergence for the bound-constrained sub-problem j**
if $E_0(x_k, z_k) \leq \omega_j$ **then** exit, solution found.
 - 3. Check convergence for the barrier sub-problem**
if $E_{\mu_{in}}(x_k, z_k) \leq \kappa_\epsilon \mu_{in}$ **then**
Update μ_{in} .
Repeat step **3** if $k=0$. Otherwise go to step **4**.
end if
 - 4. Function Evaluations**
Evaluate $c(x^k)$, $\nabla_x f(x^k)$, $\nabla_x c(x^k)$, and $\nabla^2 \mathcal{L}(x_k, \lambda' = \lambda - \mu c(x_k))$.
 - 5. Compute the search direction**
5.1 Solve (15) for d_k^x using the PCG method on GPU. .
5.2 Compute d_k^z from (11). .
 - 6. Backtracking line-search**
Calculate $\alpha_k^{x,max}$ and α_k^z based on the fraction-to-the-boundary rule.
With $\alpha_k^{x,max}$ as the initial guess, perform a line search to obtain the step size α_k^x .
 - 7. Update iteration variables and continue to next iteration**
Compute x^{k+1}, z^{k+1} with (12).
Update $k \leftarrow k + 1$.
Return to step **2**.
-

The optimality error for the barrier problems used in steps **2** and steps **3** is calculated using

$$E_{\mu_{in}}(x_k, z_k) = \max\{ \|\nabla \mathcal{L}_A - z_k\|_\infty, \|X_k z_k - \mu_{in} e\|_\infty \}. \quad (16)$$

while for step **2**, $\mu_{in}=0$.

2.6. Including General Variable Bounds

For the original problem formulation (1) considered in section 2.1, we can generalize the above algorithm and transform the barrier sub-problems to

$$\min_{x \in \mathbb{R}^n} \varphi(x) = \mathcal{L}_A - \mu_{in} \sum_{i=1}^n \ln(x^{(i)} - x_L^{(i)}) - \mu_{in} \sum_{i=1}^n \ln(x_U^{(i)} - x^{(i)}). \quad (17)$$

Coupled with this, bound multipliers for both upper bounds and lower bounds z_L and z_U are introduced. Now, the barrier term of the Hessian is defined as $\Sigma_k = S_L^{-1} Z_L + S_U^{-1} Z_U$, with $S_L = \text{diag}(x_k - x_L)$ and $S_U = \text{diag}(x_U - x_k)$.

3. Parallel implementation

The authors implemented a serial version of Algorithm 1 and Algorithm 2 in C++ and compared the runtime of different components of the implementation. The performance of the serial implementation shows that in general about 80 percent of the runtime is spent on the linear solver, 18 percent of the runtime on function evaluations, and the rest on other calculations. Therefore, an effective parallel implementation must parallelize the PCG linear solver and the function evaluations. A discussion of the parallel implementation requires a brief introduction to the GPU architecture and CUDA programming. A typical NVIDIA GPU for scientific computing contains several Streaming Multiprocessors (SMs), each of which contains several CUDA cores. The memory architecture is complex. First, the GPU has global memory accessible to all SMs. Second, each SM has its own shared memory that is accessible to all CUDA cores on this specific SM. Finally, each CUDA core has its own register memory. Global memory is typically quite large (e.g. several GB), whereas shared memory and register memory are much smaller (e.g. 48 KB and 32 KB respectively on each SM). Furthermore, global memory has high latency while shared and register memory have significantly lower latency.

In November 2006, NVIDIA introduced CUDA, a parallel computing platform and programming framework for high performance computations. A CUDA program is composed of a host program to be executed on the CPU and one or more CUDA kernels for the GPU. A kernel is executed by a grid of thread blocks, each of which can contain hundreds of threads. This can result in thousands of concurrent threads. Threads within the same block are executed by the same SM in groups of 32 threads called warps. Each SM can execute several blocks concurrently.

Different levels of software optimization techniques can be applied including memory optimization, execution configuration optimization, and instruction optimization. Some of the most important software optimization techniques are coalesced and aligned global memory access. For example, for devices that have compute capability 2.0 and support double precision data, if all threads of the same half warp access adjacent blocks of global memory that are aligned at 128-byte boundary, only one memory transaction needs to be performed by the device for all threads. Other important software optimization techniques include minimizing data transfer between the host and the device, minimizing bank conflicts in shared memory, and grid size/block size optimization (occupancy optimization). Detailed descriptions about CUDA and performance optimization are presented in the Programming Guide (NVIDIA, 2011) and the Best Practices Guide (NVIDIA, 2012) provided by NVIDIA.

3.1. Parallel PCG on the GPU

In this section, we describe the parallel GPU implementation of the linear solver for sparse systems arising from the augmented Lagrangian NLP algorithm. The main operations in the PCG method include: 1) solving the preconditioner equation, 2) vector-vector operations, and 3) matrix-vector multiplications. For a general preconditioner, one would typically need a backsolve which is, in general, inefficient when applied on a GPU. However, since we are using a diagonal preconditioner, the backsolve becomes a vector-vector operation. Vector-vector operations are straightforward to implement on the GPU. For example, to add two vectors, each element addition is performed on a separate thread. A slightly more

challenging vector-vector operation worth highlighting is the dot product because it involves a reduction operation from all CUDA cores. The details about an efficient implementation of this operation can be found in Harris (2007). Matrix-vector operations are more complex, but also have a larger scope for optimization, and we discuss these in the following sub-sections.

3.1.1. Sparse Matrix-Vector Multiplication

The most expensive step in the PCG method is the sparse matrix-vector multiplication (SpMV). The challenge when implementing a general parallel SpMV operation is that the data access is irregular for an unstructured matrix. As mentioned before, coalesced and aligned global memory access is fast while irregular global memory access is slow for a GPU. Consequently, optimizing memory access by choosing the best sparse matrix storage format for a particular matrix structure is critical for improving performance. For this purpose, we evaluate four different sparse matrix storage formats.

The coordinate (COO) format employs three arrays to store the row coordinates, column coordinates, and values of every nonzeros. Typically, nonzeros are stored row-wise, and to parallelize SpMV using the COO format, one thread is assigned for each of the nonzeros to perform the multiplication. Then the sum of threads of the same row is calculated by performing segmented reduction. The performance of SpMV with COO format is generally poor but consistent across different sparse matrix structures because of its fine granularity.

The compressed Sparse Row (CSR) format stores the values and column indices of the nonzero values in two arrays ordered row-wise. It uses a third vector to store the starting nonzero position of each row. An intuitive way of implementing SpMV for the CSR format on a GPU is to assign one thread for each of the nonzeros. However, this implementation, called a *scalar* kernel, makes the data access of contiguous threads far from each other when the number of nonzeros per row is larger than one. A more sophisticated approach used by CUSPARSE (Bell and Garland, 2009), called the *vector* kernel, assigns one warp for each row, while IBM’s SpMV library (Baskaran and Bordawekar, 2008) shows it is more efficient to assign a half warp for each row. The results of the half warps are then saved in shared memory and summed using parallel reduction. The advantage of this kernel is that the threads of the same half warp access data contiguously. Although CSR works well when the number of nonzeros in each row is larger than 16, its performance is poor when number of nonzeros per row is small, causing some threads of the half warp to remain idle.

The Ellpack (ELL) format assumes that the number of nonzeros in each row is constant and rows with fewer nonzeros are zero-padded. Since the number of nonzeros per row is fixed, it uses only two arrays to store the column indices and values of each of the nonzeros. To parallelize SpMV for the ELL format, one thread is assigned for each row. The i^{th} nonzeros in each row are stored contiguous, so the data access across contiguous threads is coalesced. However, when the number of nonzeros vary dramatically over the rows, the ELL format suffers from requiring too many zeros to pad the rows, hence it will not only cause extra computational work but also extra memory.

The Hybrid (HYB) format combines the efficiency of the ELL format for structured problems and the stability of COO format over different sparse structures. The first K entries of each row are stored in the ELL format while the remaining entries are stored in the COO format. The value of K is selected to ensure that the majority of the data is stored in the ELL parts.

Taking advantage of the discussion above about these sparse matrix formats and their implementation on the GPU, we now discuss the implementation details of PCG on GPU.

For the vector-vector operations performed in PCG, we make use of the CUBLAS library (Toolkit, 2011), which is an implementation of dense BLAS (Basic Linear Algebra Subprograms) on a GPU. However, within the PCG implementation, two results of the dot product need be transferred back from the GPU to the CPU at each iteration in order to determine whether negative curvature has been encountered and whether the PCG iterations have converged. While the data transfer for these quantities can be time consuming, the GPU used in this paper can perform data transfer between pinned host memory and global memory concurrently with device computations. We can take advantage of this by overlapping the data transfer with the SpMV kernel execution for the next step. Since the CUBLAS 4.1 library currently does not implement this overlapping feature, we have written a custom kernel for the dot product.

For SpMV, there are already several libraries to choose from. The CUSPARSE library (Bell and Garland, 2009) has implemented sparse matrix-vector multiplication with different sparse matrix formats. Recall that we do not form H_k explicitly, but rather implement the required matrix-vector products by multiplying across the expression in equation (15). For $[A_k]^T$ and $\nabla^2 \mathcal{L}$, we have selected the HYB format implemented in the CUSPARSE library since numerical results in Bell and Garland (2009) show that HYB is the fastest format for a majority of unstructured matrices. We selected the CSR format for A_k because A_k has few rows but a large number of nonzeros per row. Since the existing library does not support the overlap between data transfer and kernel execution, we have written our own CSR kernel for the SpMV.

3.2. Parallelize Function Evaluations

As we will show later in section 4, for the serial version of the algorithm the time spent on PCG iterations is around 65-85 percent of the runtime. Therefore the maximum speedup achievable throughout parallelization of the PCG steps is only about 3 to 7 times. The majority of the remaining runtime is spent on evaluating the Hessian of Lagrangian function, gradient of the objective function, gradient of the equality constraints, residual of the equality constraints, and the objective function. If these function evaluations are also parallelized, a several fold improvement can be further expected. However, so far, no library exists for automatic parallel function evaluations of general NLP problems on the GPU, and the development of a general library for this task is challenging even on the CPU. Therefore, we developed problem specific code for parallel function evaluations on the GPU for each test problem presented in our results. The purpose of further parallelizing function evaluations is to highlight the potential of this algorithm on the GPU instead of providing a general solution.

4. Numerical Results

In this section, we present the numerical performance results of the proposed algorithm on selected problems from the COPS test set. All the test problems are written in the AMPL modeling language (Gay and Kernighan, 2002). Since the augmented Lagrangian method is more efficient for problems with few constraints, the following six problems are selected -

torsion, bearing, minsurf, lane-emden, dirichlet, and henon. The first three problems have no equality constraints, while the last three problems have few equality constraints relative to the number of variables. All test problems have bound constraints on all variables. The number of variables for the selected problems ranges from 10,000 to 12,000. For comparison, the state-of-the-art nonlinear solver IPOPT with MA27 from the Harwell Subroutine Library is also used to solve these problems. For both our algorithm and IPOPT, we set the convergence tolerances for both equality constraints and optimality of the sub-problems to 10^{-6} . Both solvers are evaluated using a 2.50GHz Intel Xeon E5420 quad-core CPU and a Tesla C2050 GPU with CUDA driver version 4.2. The Tesla C2050 contains 14 SMs (each contains 32 cores) and a total of 3GB memory.

Before we parallelize the proposed algorithm, we need to make sure the serial augmented Lagrangian method is competitive when compared with existing solvers. The proposed algorithm was implemented in C++, and the timing results are compared with those from IPOPT. Figure 1 shows that although the serial algorithm is in general slower than IPOPT, the runtime ratio is below 2.5 for these test problems. Table 1 shows the wall-clock time for each of these problems.

Figure 1: Runtime of the serial augmented Lagrangian interior-point method normalized with respect to IPOPT (—).

4.1. Performance of the Parallel Code

Figure 2 shows the average performance of the parallel PCG implementation inside the augmented Lagrangian algorithm. As shown in this figure, each parallel PCG iteration performs on average 10-21 times faster than the serial implementation. Because of different architectures and round-off errors, the number of iterations taken by the parallel implementation can be different from the serial implementation. Between the two implementations,

Table 1: Wall-clock time of the serial algorithm for selected test problems.

Problem	n	m	IPOPT(s)	Serial AL-IP(s)
torsion	120000	0	21.27	21.69
bearing	120000	0	20.46	47.7
minsurf	50000	0	115.4	202.28
lane-emden	19322	81	397.67	144.54
dirichlet	18042	81	279.03	523.87
henon	10882	81	2628.87	1970.7

the number of barrier iterations never differ more than 10 percent for all test problems, and the number of PCG iterations never differ more than 60 percent. Therefore, the time per PCG iteration is used to compare the serial code and the parallel code.

Figure 2: Average speedup of each PCG iteration comparing parallel PCG implementation with serial PCG implementation (—).

Once the PCG method is parallelized, the time for function evaluations becomes the performance bottleneck. For example, Figure 3 depicts the time composition of the serial code and the parallel PCG code for test problem dirichlet. The PCG linear solver takes more than 80 percent of the total runtime in the serial code but only takes less than 30 percent in the parallel code. More than 90 percent of the remaining runtime is spent on the function evaluations, which motivates performing the function evaluations in parallel as well.

Figure 4 gives the overall speedup on all test problems with only the PCG step parallelized, and with both PCG and function evaluations parallelized. If only the PCG steps

Figure 3: Runtime composition of serial algorithm and parallel PCG algorithm for problem Dirichlet.

are parallelized, the algorithm obtains a speedup of 3-6, whereas, parallelizing the function evaluations pushes the overall speedup to 13-18.

Figure 4: Speedup with only PCG step parallelized and with both PCG and function evaluation parallelized with respect to serial implementation (—).

5. Conclusions

In summary, we have developed a parallel augmented Lagrangian interior-point algorithm for solving large-scale NLP problems using graphics processing units. The augmented Lagrangian approach ensures that the KKT system is positive definite for convex problems. This enables us to solve the KKT system using a parallel PCG method on the GPU. An overall speedup of 13-18 was obtained on six test problems from COPS test set. Three major algorithm optimizations were implemented in order to achieve these speedups. First, since each PCG iteration only requires a series of matrix-vector products with H_k , the PCG iterations were performed without explicitly forming H_k . Second, in order to ensure improved coalesced and aligned global memory access on the GPU, different matrix storage formats were utilized as appropriate. Lastly, we implemented problem specific code for parallel function evaluations on the GPU.

The augmented Lagrangian approach as implemented is best for problems with few equality constraints. Future work will explore modifications to handle more equalities. In this work we used a straightforward diagonal preconditioner, and other parallel capable preconditioners should be investigated. Finally, we manually implemented routines for parallel model evaluations, and this approach should be automated.

The GPU we use (Tesla C2050) has 448 cores, 515 Gflops double precision floating point performance, 144 GB/sec memory bandwidth and around 3 GB global memory. However, with the rapid development of parallel platforms, the latest GPUs like the Tesla K20X already have 2688 cores, 1.31 teraflops double precision floating point performance, 250 GB/sec memory bandwidth and around 6 GB global memory, which can push the performance of our GPU implementation further.

Acknowledgements

The authors would like to thank Jose Santiago Rodriguez for his comments on the algorithm and the manuscript. The authors gratefully acknowledge the financial support provided to Yankai Cao and partial financial support provided to Carl Laird by the National Science Foundation (CAREER Grant CBET# 0955205).

Reference

- Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S., Jul. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. Journal of Physics: Conference Series 180, 012037. URL <http://stacks.iop.org/1742-6596/180/i=1/a=012037?key=crossref.9bae1d50d1fbcf5b2f28c2f48f6a393d>
- Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., 2000. Multifrontal parallel distributed symmetric and unsymmetric solvers. Computer methods in applied mechanics and engineering 184 (2), 501–520.
- Baskaran, M. M., Bordawekar, R., 2008. Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies. IBM Reserach Report, RC24704 (W0812-047).

- Bell, N., Garland, M., 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, p. 18.
- Bergamaschi, L., Gondzio, J., Zilli, G., 2004. Preconditioning indefinite systems in interior point methods for optimization. *Computational Optimization and Applications* 28 (2), 149–171.
- Buatois, L., Caumon, G., Levy, B., 2009. Concurrent number cruncher: a gpu implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems* 24 (3), 205–223.
- Cao, C., Dongarra, J., Du, P., Gates, M., Luszczek, P., Tomov, S., 2013. clmagma: High performance dense linear algebra with opencl. University of Tennessee Computer Science Technical Report (Lawn 275).
- Conn, A. R., Gould, N. I., Toint, P. L., 1988. Testing a class of methods for solving minimization problems with simple bounds on the variables. *Mathematics of Computation* 50 (182), 399–430.
- Corrigan, A., Camelli, F. F., Lhner, R., Wallin, J., 2011. Running unstructured grid based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids* 66, 221–229.
- Dembo, R. S., Steihaug, T., 1983. Truncated-newton algorithms for large-scale unconstrained optimization. *Mathematical Programming* 26 (2), 190–212.
- Dick, C., Georgii, J., Westermann, R., 2011. A real-time multigrid finite hexahedra method for elasticity simulation using CUDA. *Simulation Modelling Practice and Theory* 19, 801–816.
- Dolan, E. D., Moré, J. J., Munson, T. S., 2004. Benchmarking optimization software with cops 3.0. Argonne National Laboratory Technical Report ANL/MCS-TM-273.
- Dollar, H., 2007. Constraint-style preconditioners for regularized saddle point problems. *SIAM Journal on Matrix Analysis and Applications* 29 (2), 672–684.
- Dollar, H. S., Gould, N. I., Schilders, W. H., Wathen, A. J., 2006. Implicit-factorization preconditioning and iterative solvers for regularized saddle-point systems. *SIAM Journal on Matrix Analysis and Applications* 28 (1), 170–189.
- Elble, J. M., Sahinidis, N. V., Vouzis, P., 2010. Gpu computing with kaczmarzs and other iterative algorithms for linear systems. *Parallel computing* 36 (5), 215–231.
- Forsgren, A., Gill, P. E., Griffin, J. D., 2007. Iterative solution of augmented systems arising in interior methods. *SIAM Journal on Optimization* 18 (2), 666–690.
- Galoppo, N., Govindaraju, N. K., Henson, M., Manocha, D., 2005. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In: *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, p. 3.

- Gay, D. M., Kernighan, B., 2002. *Ampl: A modeling language for mathematical programming*. Duxbury Press/Brooks/Cole 2.
- Golub, G. H., Van Loan, C. F., 2012. *Matrix computations*. Vol. 3. JHU Press.
- Gould, N. I., Hribar, M. E., Nocedal, J., 2001. On the solution of equality constrained quadratic programming problems arising in optimization. *SIAM Journal on Scientific Computing* 23 (4), 1376–1395.
- Harris, M., 2007. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology* 6.
- Helfenstein, R., Koko, J., 2011. Parallel preconditioned conjugate gradient algorithm on gpu. *Journal of Computational and Applied Mathematics*.
- Hogg, J., Scott, J., 2010. An indefinite sparse direct solver for large problems on multicore machines.
- Jr., F. M., Szakly, T., Mszros, R., Lagzi, I., 2010. Air pollution modeling using a Graphics processing united with CUDA. *Computational Physics Communications* 181, 105–112.
- Krawezik, G. P., Poole, G., 2010. Accelerating the ansys direct sparse solver with gpus. 2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC10).
- Kumbhar, P., 2011. Performance of petsc gpu implementation with sparse matrix storage schemes. Ph.D. thesis, MSc Thesis, University of Edinburgh, Edinburgh: University of Edinburgh.
- Li, R., Saad, Y., 2013. Gpu-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing* 63 (2), 443–466.
- Lin, C.-J., Moré, J. J., 1999. Newton’s method for large bound-constrained optimization problems. *SIAM Journal on Optimization* 9 (4), 1100–1127.
- Lucas, R. F., Wagenbreth, G., Tran, J. J., Davis, D. M., 2012. Multifrontal sparse matrix factorization on graphics processing units. Information Sciences Institute, University of Southern California, Tech. Rep.
- Lukšan, L., Vlček, J., 1998. Indefinitely preconditioned inexact newton method for large sparse equality constrained nonlinear programming problems. *Numerical linear algebra with applications* 5 (3), 219–247.
- Naumov, M., 2011. Incomplete-lu and cholesky preconditioned iterative methods using cusparse and cublas. Nvidia white paper.
- Nocedal, J., Wright, S. J., 2006. *Numerical optimization*. Springer Science+ Business Media.
- NVIDIA, 2011. *CUDA Programming Guide*, Version 4.1.
- NVIDIA, 2012. *CUDA C Best Practices Guide*, Version 4.1.

- Perugia, I., Simoncini, V., 2000. Block-diagonal and indefinite symmetric preconditioners for mixed finite element formulations. *Numerical linear algebra with applications* 7 (7-8), 585–616.
- Roman, G., Joldes, Adam Wittek, K. M., 2010. Real-time nonlinear finite element computations on GPU Application to neurosurgical simulation. *Computer Methods in Applied Mechanics and Engineering* 199, 3305–3314.
- Schenk, O., Gärtner, K., 2004. Solving unsymmetric sparse systems of linear equations with pardiso. *Future Generation Computer Systems* 20 (3), 475–487.
- Tomov, S., Nath, R., Ltaief, H., Dongarra, J., Apr. 2010. Dense linear algebra solvers for multicore with GPU accelerators. 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 1–8.
URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5470941>
- Toolkit, C., 2011. 4.0 cublas library. NVIDIA Corporation.
- Vouzis, P. D., Sahinidis, N. V., 2011. Gpu-blast: using graphics processors to accelerate protein sequence alignment. *Bioinformatics* 27 (2), 182–188.
- Wächter, A., Biegler, L. T., 2006. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming* 106 (1), 25–57.
- Yeralan, S. N., Davis, T., Ranka, S., 2013. Sparse QR factorization on GPU architectures. Tech. rep., Technical report, University of Florida (November 2013).
- Zhang, Y., Vouzis, P., Sahinidis, N. V., 2011. Gpu simulations for risk assessment in co 2 geologic sequestration. *Computers & Chemical Engineering* 35 (8), 1631–1644.