Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 495–504

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems

Jack Dongarra[1,2], Sven Hammarling[2], Nicholas J. Higham[2], Samuel D. Relton[2], Pedro Valero-Lara[3], and Mawussi Zounon[2]

[1] University of Tennessee, Oak Ridge National Laboratory, TN, USA.
dongarra@icl.utk.edu
[2] School of Mathematics, The University of Manchester, Manchester, UK.
{nick.higham, samuel.relton, mawussi.zounon}@manchester.ac.uk
[3] Barcelona Supercomputing Center, Barcelona, Spain.
pedro.valero@bsc.es

Abstract

A current trend in high-performance computing is to decompose a large linear algebra problem into batches containing thousands of smaller problems, that can be solved independently, before collating the results. To standardize the interface to these routines, the community is developing an extension to the BLAS standard (the batched BLAS), enabling users to perform thousands of small BLAS operations in parallel whilst making efficient use of their hardware. We discuss the benefits and drawbacks of the current batched BLAS proposals and perform a number of experiments, focusing on a general matrix-matrix multiplication (GEMM), to explore their affect on the performance. In particular we analyze the effect of novel data layouts which, for example, interleave the matrices in memory to aid vectorization and prefetching of data. Utilizing these modifications our code outperforms both MKL[1] and CuBLAS[2] by up to 6 times on the self-hosted Intel KNL (codenamed Knights Landing) and Kepler GPU architectures, for large numbers of double precision GEMM operations using matrices of size 2×2 to 20×20.

Keywords: BLAS; Batched BLAS; Scientific computing; High-performance computing; Memory management; Parallel processing

1 Introduction

Over the past few decades there has been a tremendous amount of community effort targeting the design and implementation of efficient linear algebra software. The main focus of this drive has been to solve larger problems in less time. As a result, numerous libraries have been designed to take advantage of advances in computer architecture and exploit the parallelism

[1]https://software.intel.com/en-us/intel-mkl
[2]http://docs.nvidia.com/cuda/cublas

both within a single node (using hardware accelerators), and between nodes (communicating using MPI, for example).

In an attempt to utilize highly parallel computing resources more efficiently, there is a current trend towards splitting large linear algebra problems into thousands of smaller subproblems that can be solved concurrently [2]. One example of this is given by multifrontal solvers for sparse linear systems [7]. Many popular linear algebra libraries such as Intel MKL and NVIDIA CuBLAS have begun to provide limited support for this approach but no complete set of linear algebra routines operating on batches of small matrices is available.

The solution to this problem is to develop a new standard set of routines for carrying out linear algebra operations on batches of small matrices, building on the well-known Basic Linear Algebra Subproblems (BLAS) standard [5], [6], [10]. The idea behind the batched BLAS (BBLAS) is to perform multiple BLAS operations in parallel on many small matrices, making more efficient use of the hardware than a simple OpenMP for loop would allow. For example, if we consider a general matrix-matrix multiplication (GEMM) operation over a batch of N matrices then we would like to compute, in parallel,

$$C_i \leftarrow \alpha_i A_i B_i + \beta_i C_i, \qquad i = 1 : N. \tag{1}$$

In this example we might keep the sizes of the matrices and the values of α_i and β_i constant throughout the entire batch or allow them to vary, depending upon the application that we have in mind.

One particular application which can benefit dramatically from performing many small matrix multiplications in parallel is deep learning: the batched GEMM functionality in vendor libraries is already being utilized in popular machine learning libraries such as TensorFlow [1] and Theano [4]. Further examples of applications where the solution of many small problems are required include domain decomposition [3], the rendering of 3D graphics in web browsers [8], metabolic networks [9], astrophysics [12], matrix-free finite element methods [11], and the solution of separable elliptic equations [14].

Currently, libraries that implement BBLAS functionality use a relatively simple memory layout (explained in section 2) which generally gives suboptimal performance. One of our primary goals in this paper is to investigate a number of potential optimizations to increase the performance of BBLAS routines for small matrices on modern parallel architectures. We explore, amongst other things, the effect of different API designs and memory layouts on performance. Currently, there is no standard interface for BBLAS operations and no complete implementation of batched BLAS routines is available. Intel MKL has support for batched GEMM computation whilst NVIDIA CuBLAS supports batched GEMM and triangular solve (TRSM), plus some batched LAPACK routines; but these libraries do not share the same API.

The remainder of this article is organized as follows. In section 2 we outline the different APIs for BBLAS and perform some experiments to compare their associated overheads. Section 3 contains discussion and experiments to determine the effect that the memory layout has on the performance of BBLAS operations and the transfer to and from hardware accelerators, which are an important consideration when designing an efficient API. In section 4 we focus on the performance of a novel data layout, which interleaves the batches of matrices in memory, on both GPUs and the self-hosted Intel KNL. Concluding remarks are given in section 5.

2 Batched BLAS

Two main approaches can be taken to allocate computational resources to batched BLAS operations. First, we could compute each BLAS operation in order and allocate all available cores