Chapter 5
Batched Computing

ABSTRACT

This chapter presents the concept of batched computing, which consists of the division of a large problem into smaller portions and can be applied to both dense and sparse linear algebra. Two examples, general matrix-matrix multiplication (GEMM) and general triangular solver (GTSV), are used to present different approaches depending on the problem to solve. The GEMM example focuses on multi-core platforms, and it is also used to introduce the concept of auto-tunning. In the case of GTSV, the targeted device is a GPU. Moreover, given that this is a sparse operation, an analysis of the data layout is presented to see the impact of this aspect.

INTRODUCTION TO BATCH COMPUTING

Many scientific applications rely on high performance computing libraries such as PLASMA, Intel MKL, NVIDIA cuBLAS, or hipBLAS to accelerate their linear algebra operations. In some fields, such as fluid dynamics, econometrics, control theory, or computational statistics, medium to large dense matrices are processed, so techniques like those presented in Chapter 3 may suffice to quickly obtain the results. On the other hand, there are numerous applications that belong to the astrophysics (Messer et al., 2013), hydrodynamics (Dong et al., 2014), image/signal processing areas (Anderson et al., 2012) (Molero et al., 2013), numerical ocean models (Halliwell, 2004), or the simulation of the human brain (Valero-Lara, Martinez-Perez et al., 2017), among others, that usually require sparse linear algebra computations. In those cases, the

DOI: 10.4018/978-1-7998-7082-1.ch005

overall computation can be cast in terms of many small problems (that fit into specific cache levels) that can be treated as either DLA or SLA operations, that is a set (batch) of independent DLA/SLA operations.

In this scenario, one can think about batches of fixed and variable size. In the former all the problems to be solved within the batch have the same size, as in (Dongarra et al., 2017), (Valero-Lara, Martinez-Perez et al., 2017), (Valero-Lara et al., 2016) or (Valero-Lara, Martínez-Pérez et al., 2018). While in the latter, each task may have a different size, as targeted in (Abdelfattah et al., 2016). This chapter is focused on batched operations on multicore and GPU platforms. The first part of the chapter is focused on variable size DLA batches, since the approach can be seen as a generalization of the fixed size problem. More specifically, the first part of the chapter analyzes the performance of different implementations for the batched GEMM kernel on a multi-core architecture. The Intel MKL cblas_dgemm_batch (Zhao, 2017) is included as a reference. Note that in this case, no implementation details are available. Moreover, six more approaches are analyzed; two of them make use of parallel for schedulers, similarly to that proposed in (Abdelfattah et al., 2016). The other four strategies are task-based implementations that rely on different mechanisms to achieve a better workload distribution. Given that some of the task-based approaches need to be tuned to attain high performance, auto-tuning strategies are also tackled at the end of the first part of the chapter.

The second part of the chapter focuses on SLA batches, both fixed and variable sizes. In that part, a batch of tridiagonal systems on GPUs are tackled. In this case, given that the tridiagonal system solver is already a sparse operation (see Chapter 4), the chapter focuses on both fixed size batches, where the sparsity is targeted at the level of each problem in the batch, and variable size batches, where an extra level of imbalance is introduced due to the variability of the size of each problem. More specifically, this part of the chapter analyzes the performance of two GPU Thomas algorithm implementations (fixed and variable size). The gtsvStrideBatch routine from cuSPARSE NVIDIA (NVIDIA) is included as a reference.

In the following section, the authors describe how to implement batched GEMM routines and how to leverage fork-join as well as task based parallel approaches to attain an optimal parallel performance.