Boyana Norris

Automatic Kernel Acceleration of PETSc Krylov Solvers

MCS Division
Bldg 240
Rm 2152
Argonne National Laboratory
9700 South Cass Avenue
Argonne
IL 60439
norris@mcs.anl.gov
Cherkuri Choudary
Deepan Balasubramanian
Jeswin Godwin
Daniel Lowell
Azamat Mametjanov
Boyana Norris
P. Sadayappan, Gerald Sabin, Sravya Tirukkovalur

In the last few years, the use of accelerator architecture in both desktop and high-performance platforms has been rapidly increasing. Crays XK6 systems have a hybrid architecture using NVIDIA Graphics Processing Units (GPUs), and in a similar spirit, Intel has created the Many Integrated Core (MIC) architecture codenamed Knights Corner. Developing software for such heterogeneous architectures has become increasingly important, but requires significant expertise and development effort. Furthermore, the code developed and optimized for one accelerator or MIC architecture is generally not easily portable to another. One approach that addresses this challenge is to define code transformations that enable the same reference implementation of an algorithm to be ported to multiple platforms. An extension of this approach is to enable users to express an algorithm in a high-level language, which can then be transformed automatically into multiple low-level source code representations for execution on different accelerator architectures. Examples of low-level target codes include C with OpenMP directives or CUDA C kernels.

Krylov subspace solvers are widely used iterative methods for solving large, sparse, linear systems of equations, with implementations provided by a number of popular numerical toolkits such as the Portable Extensible Toolkit for Scientific Computation (PETSc). PETScs open-source, extensible, and modular framework allows computational scientists or other library developers to create new or modify existing solvers as needed.

The implementation of a Krylov solver for execution on a GPU platform takes place within the framework of PETSc vector and matrix data types. PETSc has several GPU implementations of iterative solvers such as the Generalized Minimum Residual Method (GMRES). These implementations use PETScs vector and matrix data structures, but implement the algorithms by calling GPU libraries external to PETSc. These libraries are provided by different CUDA toolkits (e.g., the CUSP and THRUST libraries). The delegation of control to GPU libraries provides a modular design which hides the lower level implementations on the GPU, leaving the algorithm developer free to use the familiar high-level operations. This black-box approach is often suboptimal because the presence of multiple library calls hampers locality-exploiting optimizations of low-level kernels. Our transparent implementation of vector and matrix types keep intact the modular design of PETSc while allowing the developer full access to the tuning of low-level operations.

We explored a number of optimizations in our vector and matrix data structures, including GPU global memory alignment, reduction of data transfers between the host and the device, and maximizing GPU utilization. Examples of these optimizations include asynchronous streaming kernels and memory calls, pipelined reductions of large array operations, use of dynamically allocated shared memory, register offloading of global memory, and pinned host memory. The goal of this exploratory step was to create the data structures and support functions that can serve as templates for automated code generation and tuning of the core kernel operations of a Krylov iterative solver.

After obtaining kernel templates, we began developing code transformations of C-based implementations to CUDA C-based optimized versions. These transformations were developed within the optimizing autotuner Orio. Orio is a recently developed, extensible, and portable software generation framework for empirical performance tuning. It takes annotated C or Fortran source code as input, performs different performance-tuning transformations on the internal code representation, generates multiple code variants corresponding to the different transformations, empirically evaluates the performance of the generated codes, and selects the best-performing variant using several popular heuristic search algorithms. Orio annotations consist of semantic comments that specify the computation and optionally, the transformations to perform. A separate tuning specification contains various parametrized performance-tuning directives and bounds of optimization space to search. In addition to the general-purpose tuning directives, such as loop fusion and unrolling, tiling, and scalar replacement, Orio supports a number of architecture-specific optimizations (e.g., generating calls to SIMD intrinsics on Intel and Blue Gene/P architectures). Orio is implemented in Python and can be dynamically extended by placing new language modules (containing implementations of parsing, transformation, and code generation interfaces) into the canonical path. Typically, most of the execution time is spent in loops; hence, we have extended Orios Loop module, which provides a C-like syntax for specifying array-based computations. The extension parses user-provided GPU transformation directives and generates CUDA kernels for execution on GPU devices. The annotated loops are replaced by CUDA-specific resource marshaling for parallel execution (e.g., data transfer, memory allocation, and thread layout) and unmarshaling upon the conclusion of the GPU computation.

The proposed generation of kernels allows us to create a testbed for generating and tuning iterative solver implementations and testing them within the PETSc framework, including benchmarking the performance of new GPU implementations against existing ones. We will present a detailed comparison of the performance of our approach and existing manually-tuned and library-based PETSc CPU and GPU implementations.

Future work includes enabling the developer to use vectors and (sparse) matrices as base types and support code generation and tuning for a core set of linear algebra operations. For example, instead of explicitly iterating over index vector elements in, say an AXPY operation, one can specify just y = y + alpha * x. Other future work includes implementing more complex kernel code generation, integrating more aggressive optimizations into the generated kernel variants, and accelerating other solvers.