A General Framework for Multicore Programming with VSIPL++

Brooks Moses, Jules Bergmann, Stefan Seefeld, Don McCoy, Mike LeBlanc
CodeSourcery, Inc.
{brooks, jules, stefan, don, mike}@codesourcery.com

Introduction

High-performance programming for multicore architectures introduces complex challenges in parallelizing algorithms and managing data movement between cores. Libraries such as Sourcery VSIPL++ [1] are useful in cases where the program can be expressed using standard array operations such as FFTs and linear algebra, as the parallelism can be encapsulated within the library. However, challenges remain when the programs cannot be written in terms of standard operations and the parallelism must be exposed to the programmer.

Streaming has emerged as a powerful paradigm for addressing this situation. Breaking large computations into smaller blocks that can be processed nearly independently reduces problems to sizes that fit into a single core's memory, and provides a framework in which computation and communication can be effectively overlapped.

Simple streaming problems are straightforward to analyze and describe. However, in real applications, streaming must often deal with complex data dependencies such as those that occur with fused streams or non-linear access patterns. In this paper, we present a framework for describing and optimizing stream computations in terms of array blocks, which we will be incorporating into Sourcery VSIPL++. Using the open-standard VSIPL++ API [2] allows streaming problems to be described portably at a high level, and the Sourcery VSIPL++ implementation dispatch framework is able to execute the resulting program efficiently to achieve high performance.

Generalizing Multicore Programming

Many parallel operations on arrays can be expressed in a block form: the arrays are divided into blocks, and then "kernel" operations are applied to these blocks. These operations are unordered aside from dependency chains, which are typically short compared to the number of operations to be performed, but may have arbitrary complexity. Much of the difficulty in implementing these operations on multicore architectures is determining on which core and in what order each of these operations should be run, and managing the transfers of the data blocks to and from the cores.

These are general challenges. Although the shape of the dependency tree and the number and capabilities of the available cores vary, the optimization problem is amenable to general architecture- and problem-independent solution algorithms.[1] Similarly, although the code for managing the data transfers differs from architecture to architecture, it is a problem-independent process and amenable to a general solution for each architecture.

The framework we are currently developing supplies a general mechanism for ordering kernel-operation sets into lists of operations to be performed by each core, and architecture-specific mechanisms for dispatching data blocks to the cores and executing kernel functions on them. Thus, the only work required to implement a particular array operation is to implement the kernel functions for the relevant cores and specify the unordered operation set to be performed.

Benefits of Using Sourcery VSIPL++

Because this framework is being developed as an extension to Sourcery VSIPL++, it can take advantage of the existing VSIPL++ high-level API and Sourcery VSIPL++'s high-performance implementation techniques.

In VSIPL++, multidimensional array data is stored in vector, matrix, and tensor objects that present a consistent representation to the programmer, independent of how the data is stored in memory. Thus, an implementation can alter the data storage – for instance, by distributing an array across the local memory of several cores, or by reordering the data into a "blocked" format that allows for faster transfers – without affecting the interface by which that data is accessed in top-level programs.

Sourcery VSIPL++ includes a dispatch mechanism by which an operation on array data can be dispatched to multiple implementations at compile time or at runtime depending on various parameters of the input data. For example, an operation could be dispatched to a parallel implementation for large problem sizes, but to a serial implementation for small problem sizes where the startup costs would swamp the benefits from parallelization.

Finally, most parallel operations are performed in a context of a larger program, and using the VSIPL++ API allows data to be easily transferred between user-written parallel operations and Sourcery VSIPL++ library functions.

Framework Architecture

There are three points at which programmers interact with this framework. Parallel operations are defined by specifying a set of "kernel" functions to be applied to array blocks, and a description of the set of array blocks to which these kernels are applied. Once defined, the parallel operation takes the form of a function that can then be applied to VSIPL++ vector, matrix, or tensor objects.

When a programmer uses this function call, the dispatch mechanism invokes the first layer of the framework: the Operation Set Generator. This uses the dimensions of the input arrays and the description of the set of blocks to which the kernels are to be applied, and produces a set of kernel operations and an associated dependency tree.

This set of kernel operations is then passed to the Scheduler. The Scheduler determines the order in which

[1] This is much like the problem of ordering instructions in a multiple-architecture optimizing compiler such as GCC, and can be addressed by similar mechanisms.

the kernel operations are to be executed on each of the available cores and translates them into a sequence of instructions. These instructions consist of data transfer operations and calls to the kernel functions, and are represented in a compact architecture-independent bytecode format that can be passed to the individual cores.

The Dispatch Engine then transfers these bytecoded instruction sequences to a lightweight execution framework on each core, which executes them.

Preliminary Results

At the time of this writing, we have implemented a proof-of-concept demonstration of this framework for a simple elementwise matrix addition and obtained performance data on three different multicore architectures. It should be noted that these are preliminary results; the framework has not yet been optimized for performance at all.

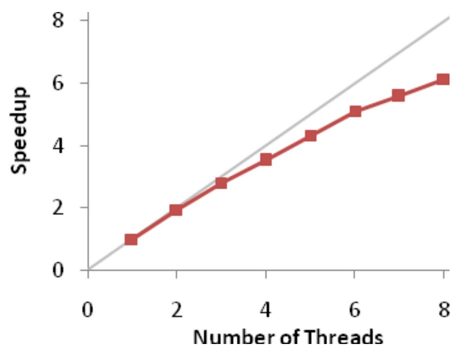

Figure 1: Two Intel Xeon processors with 4 x86-64 cores per processor, based on pthreads.

Figure 1 shows the parallel speedup results on a 2-processor machine with 4-core Xeon processors, using an implementation based on the POSIX pthreads library. When all eight cores are in use, a speedup factor of 6.13 is obtained compared to the performance on a single core, for a parallel efficiency of 76%.

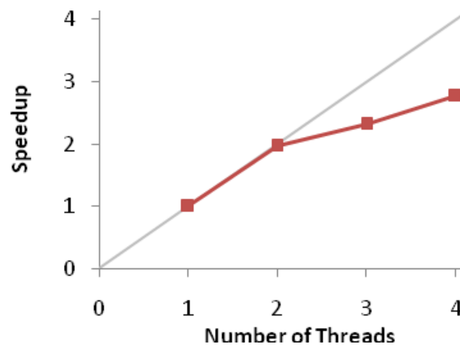

Figure 2: Two IBM Cell processors using 1 dual-threaded Power core per processor, based on pthreads.

Figure 2 shows the equivalent results for the Power-architecture PPU cores on the two Cell processors on an IBM QS21 Cell Blade, again with a pthreads-based implementation. Each Cell processor has a single PPU core, with two hardware threads per core which share the core's execution units. Thus, there is a near-linear speedup (a factor of 1.96) when going from one thread to two threads, and more modest improvements for three or four threads; the speedup with four threads is a factor of 2.76.

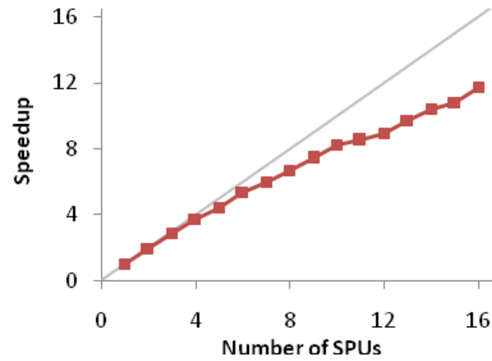

Figure 3: Two IBM Cell processors using 8 SPU cores per processor, based on ALF.

Figure 3 shows results for the SPU cores (8 per processor) on the same two-processor IBM Cell Blade, using an implementation based on IBM's ALF library. With all sixteen SPUs in use, a speedup factor of 11.7 was obtained compared to the performance on a single SPU, for a parallel efficiency of 73%.

Conclusion

A large class of array computations can be represented with block operations. We are producing an extension to Sourcery VSIPL++ which enables programmers to implement these computations on multicore platforms by writing kernel functions for the block operations and a descriptor for the set of blocks operations to perform, with the library handling scheduling and execution of these operations and the associated data transfers on the various processor cores.

The preliminary performance data are encouraging, and indicate that this framework can be expected to produce performance comparable to that obtained by hand-coding, while simplifying the programming process and enhancing portability to new architectures.

We would also expect that, for operations with more complex data dependencies, an automated scheduling algorithm could outperform the average hand-coded program, just as optimizing compilers can outperform all but the best handwritten assembly code.

References

[1] CodeSourcery, Inc. Sourcery VSIPL++. [online] Available: http://www.codesourcery.com/vsiplplusplus.

[2] CodeSourcery, Inc. *VSIPL++ Parallel Specification 1.0.* Georgia Tech Res. Corp. 2005 [online] Available: http://www.hpec-si.org.