A Collection-Oriented Programming Model for Performance Portability

Saurav Muralidharan
University of Utah
Salt Lake City, UT, USA
sauravm@cs.utah.edu

Michael Garland
NVIDIA Corporation
Santa Clara, CA, USA
mgarland@nvidia.com

Bryan Catanzaro
Baidu Inc.
Sunnyvale, CA, USA
bcatanzaro@acm.org

Albert Sidelnik
NVIDIA Corporation
Santa Clara, CA, USA
asidelnik@nvidia.com

Mary Hall
University of Utah
Salt Lake City, UT, USA
mhall@cs.utah.edu

Abstract

This paper describes Surge, a collection-oriented programming model that enables programmers to compose parallel computations using nested high-level data collections and operators. Surge exposes a code generation interface, decoupled from the core computation, that enables programmers and autotuners to easily generate multiple implementations of the same computation on various parallel architectures such as multi-core CPUs and GPUs. By decoupling computations from architecture-specific implementation, programmers can target multiple architectures more easily, and generate a search space that facilitates optimization and customization for specific architectures. We express in Surge four real-world benchmarks from domains such as sparse linear-algebra and machine learning and from the same performance-portable specification, generate OpenMP and CUDA C++ implementations. Surge generates efficient, scalable code which achieves up to 1.32x speedup over handcrafted, well-optimized CUDA code.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.4 [*Processors*]: Code generation; D.3.2 [*Language Classifications*]: Concurrent, distributed and parallel languages

Keywords Nested-data-parallelism, performance-portability

1. Introduction

The recent proliferation of platforms that combine latency-optimized general-purpose multicore parallel architectures with throughput-optimized accelerators such as GPUs may employ different parallel programming models (e.g., OpenMP and CUDA) on the same machine. Proposed extensions to OpenMP may reduce this programming burden through a common programming model, but nevertheless code developed for one architectural paradigm is difficult to port to a completely different paradigm. Expert programmers, who are familiar with lower-level details of the target architecture, commonly produce software customized to its specific features. The resulting low-level code is difficult to maintain and likely not portable to future architectures. In contrast, domain experts, who may not be well-versed with the details of the target architecture, are often looking for high-level abstractions for parallelism that also help them obtain good performance on current and future parallel architectures. To satisfy both groups, parallel programming models that provide both (1) high-level abstractions, and (2) performance portability, i.e., enable construction of programs that perform well across a variety of current and future parallel architectures, are thus critically important. This paper describes a new collection-oriented parallel programming model called *Surge*. Surge supports nested data-parallel collections and operations as in models such as NESL [2], but decouples the specification of computations from architecture-specific implementation details using *schedules*. A schedule specifies how a computation is mapped onto the target hardware platform. This decoupling is one of the unique features of Surge and it allows expert programmers to (1) target multiple architectures more easily, and (2) generate a search space of possible implementations, from among which optimal ones can be found either manually or using techniques such as autotuning. Additionally, a schedule inference mechanism helps find good implementations automatically to enable non-expert programmers to use the system to write performance-portable programs.

2. Programming Model Overview

Surge exposes a number of collections and data-parallel primitives (also called operators) which can be nested within each other to express individual computations. Consider the sparse matrix-vector multiply (SpMV) example from Listing 1. This example uses high-level collections and operators to express the core computation in about six lines of code (excluding comments). From this concise specification, Surge is capable of generating multiple CPU and GPU implementations, with some of the GPU implementations outperforming the hand-tuned CUSP library [1].

2.1 Operators and Collections

Collection-oriented programming models provide primitives that operate on collections of data. Surge provides a number of high-

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
ACM 978-1-4503-3205-7/15/02
http://dx.doi.org/10.1145/2688500.2688537

```
auto inner_product =
  [=](S row, I indices) {
    // Gather elements from vector x
    auto z = gather(x, indices);

    // Perform element-wise
    // multiplication of x with row
    auto vector_mul = map(mul, row, z);

    // Obtain inner product by
    // summing up elements of vector_mul
    return reduce(plus, vector_mul);
  }
// Apply inner product over all rows of matrix
y = map(inner_product, s_matrix, s_indices);
```

Listing 1: Surge code for SpMV

level operators inspired by existing data-parallel programming models such as NESL and Copperhead [3], and provides collections using a `sequence` type, which represents a view over the underlying data. A sequence of length n is a collection indexed by contiguous integers $[0, n)$, and is defined in our current implementation using a pair of iterators pointing to the start and end of the underlying data. Surge also supports multi-dimensional arrays that are analogous to sequences, but present a multi-dimensional view over the underlying (flat) data instead of a one-dimensional view. Examples of some core data-parallel operators in Surge are: `map, reduce, scan` etc. In addition to these, Surge also provides a set of *sequence operators* that create and transform sequences. Examples of sequence operators include `transform, range, split` etc.

In the SpMV example, `x` and `y` are flat sequences, and `s_matrix` and `s_indices` are nested sequences represented in the compressed sparse row (CSR) format. Such non-uniform sequences contain sub-sequences representing individual tiles or rows of the original sequence and can be obtained using the `nest` sequence operator. In the example, the `row` and `indices` arguments denote these sub-sequences. `S` and `I` are defined as `decltype(s_matrix[0])` and `decltype(s_indices[0])`, respectively.

2.2 Schedules

For each computation expressed in Surge, a schedule and platform may be optionally specified. Surge currently generates code for two hardware platforms: GPUs and x86 CPUs through CUDA C++ and OpenMP, respectively. Each platform has an associated set of data-parallel *kernels* that implement various $\langle$operator, schedule$\rangle$ combinations on that platform. In our C++ implementation, platforms are denoted as user-defined types.

Surge currently provides three primitive schedules: `independent` (for operators with fully independent iterations such as `map`), `cooperative` (for operators that require a *coordination phase* among participating threads such as `reduce` and `scan`), and `sequential` (for operators with an enforced ordering) that provide a basis for composing more complex schedules. Similar to platforms, schedules are represented as user-defined types in our current implementation. Schedules may be nested to correspond with operator nesting. We use the <> symbols to denote nesting. For example, `independent<cooperative>` denotes a nested schedule that evaluates the outer operator using the `independent` schedule and the inner operator using the `cooperative` schedule.

As mentioned in Section 1, Surge features a schedule inference system that can automatically find high-quality schedules for nested operators. Schedule inference proceeds along two phases: (1) *schedule construction*, and (2) *platform-aware schedule transformation*. In phase one, a machine-independent schedule is constructed based on the nesting structure of operators. In phase two, this machine-independent schedule is transformed into a platform-specific schedule. For the SpMV example in Listing 1, the default inferred schedule is `independent<cooperative>` in phase one and `independent_warps<cooperative_warp>` in phase two. The latter CUDA-specific schedule assigns a warp for the processing of each matrix row in the computation. Since schedules are represented as static types in our current implementation, C++ partial template specialization is used to implement both these phases in the type system.

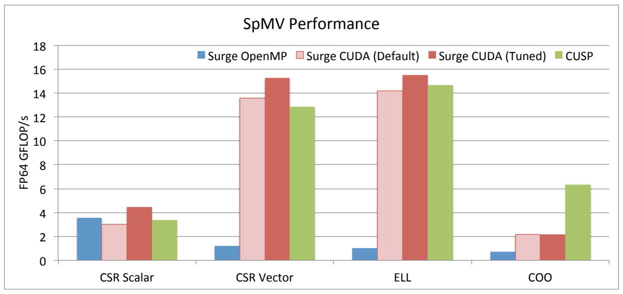

Figure 1: SpMV performance

3. Evaluation

We evaluate our benchmarks on a system with an Intel Core i7 3820 CPU (4-core, 8 logical threads with Hyper-Threading) with 8 GB of RAM. The GPU used is an NVIDIA K20c (Kepler). We use the NVIDIA CUDA compiler (NVCC) 6.5 (with the g++-4.8.2 host compiler) with the `-O3` flag.

Figure 1 shows performance results for our SpMV benchmark. We evaluate the performance of four SpMV implementations generated by Surge from the same specification (Listing 1) on two different platforms: CPU (OpenMP) and GPU (CUDA). We compare their performance with the corresponding GPU reference implementations from the CUSP library [1]. Here, the *tuned* bars represent performance achieved by manually finding optimal values for the tunable parameters exposed by Surge, while the *default* bars represent performance achieved by using the default values for these parameters. By default, the CSR Vector variant is inferred by Surge for the SpMV computation.

In addition to SpMV, we implemented three other benchmarks in Surge and evaluated their performance. The benchmarks are: (1) k-Means clustering, (2) support vector machine (SVM) training, and (3) global prefix scan. Our benchmarks achieved speedups of up to 3.79x on CPU (over corresponding reference sequential implementations) and up to 1.32x on GPU (over corresponding hand-crafted reference GPU implementations) on large input sizes. Further, to provide a rough measure of productivity, we counted the lines of code required in Surge to express the core computations of our benchmarks and compared it with the number of lines taken to express the same computation in our reference GPU implementations. We found that computations expressed in Surge require, on average, 7.3x less lines of code than optimized CUDA versions.

References

[1] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proc. Conference on High Performance Computing Networking, Storage and Analysis*, Nov. 2009.

[2] G. Blelloch. Nesl: A nested data-parallel language. Technical report, Carnegie Mellon University, 1992.

[3] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 47–56. ACM, 2011.