Designing a Tunable Nested Data-Parallel Programming System

SAURAV MURALIDHARAN, University of Utah
MICHAEL GARLAND and ALBERT SIDELNIK, NVIDIA Corporation
MARY HALL, University of Utah

This article describes Surge, a nested data-parallel programming system designed to simplify the porting and tuning of parallel applications to multiple target architectures. Surge decouples high-level specification of computations, expressed using a C++ programming interface, from low-level implementation details using two first-class constructs: schedules and policies. Schedules describe the valid ways in which data-parallel operators may be implemented, while policies encapsulate a set of parameters that govern platform-specific code generation. These two mechanisms are used to implement a code generation system that analyzes computations and automatically generates a search space of valid platform-specific implementations. An input and architecture-adaptive autotuning system then explores this search space to find optimized implementations. We express in Surge five real-world benchmarks from domains such as machine learning and sparse linear algebra and from the high-level specifications, Surge automatically generates CPU and GPU implementations that perform on par with or better than manually optimized versions.

CCS Concepts: • **Software and its engineering** → *Software performance*; *Parallel programming languages;*

Additional Key Words and Phrases: Nested data parallelism, autotuning, performance portability

ACM Reference Format:
Saurav Muralidharan, Michael Garland, Albert Sidelnik, and Mary Hall. 2016. Designing a tunable nested data-parallel programming system. ACM Trans. Archit. Code Optim. 13, 4, Article 47 (December 2016), 24 pages.
DOI: http://dx.doi.org/10.1145/3012011

1. INTRODUCTION

Parallel architectures are becoming increasingly diverse. With high-performance code being commonly customized to specific low-level architectural features, the cost of developing and maintaining performance-portable applications is also getting higher. In this article, we introduce a new programming system, named *Surge*, which supports decoupling the high-level specification of computations from their implementation details using first class constructs. This separation enables Surge to easily generate a *search space* of multiple low-level, architecture-specific implementations from the same specification. Expert users or automatic performance tuning systems (autotuners) are

This is a new article, not an extension of a conference article.
This research was funded by DARPA contract HR0011-13-3-0001. This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.
Authors' addresses: S. Muralidharan, M. Garland, and A. Sidelnik, NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050; emails: {sauravm, mgarland, asidelnik}@nvidia.com; M. Hall, School of Computing, University of Utah, 201 Presidents Circle Rm 201, Salt Lake City, UT 84112; email: mhall@cs.utah.edu.

DOI: http://dx.doi.org/10.1145/3012011

then able to navigate this search space to find the best implementation for a given execution context (architecture and input dataset).

Surge consists of a programming interface, implemented purely as a C++ library, and separate code generation and autotuning subsystems. The programming interface is based on *nested data parallelism*, which is a generalization of flat data parallelism; in nested data parallelism, subcomputations of a data-parallel computation may themselves be data-parallel [Blelloch 1992]. It is a powerful abstraction for expressing a variety of parallel computations; further, the algorithmic hierarchy in nested data-parallelism maps naturally to modern processors, many of which have hierarchically organized execution and storage resources (such as GPUs).

The separation between nested data-parallel specification and implementation is achieved using two constructs in the programming interface: *schedules* and *policies*. These represent different levels of abstraction with respect to code generation: Schedules characterize the dependence structure of data-parallel operators, independent of hardware-specific details, while policies encapsulate a set of *optimization parameters* that govern low-level code generation on various hardware platforms. This two-level design makes targeting new platforms easier and provides a systematic way of automatically generating a search space of valid implementations, which we then navigate using an autotuner. The resulting system is easy to use for application developers but still provides performance that is on par with or better than manually optimized implementations for a variety of computations.

While there exist other nested data-parallel programming systems, most automatically employ flattening and then rely on flat data-parallel mappings [Blelloch 1992; Chakravarty et al. 2007; Bergstrom et al. 2013]. Copperhead [Catanzaro et al. 2011] and CuNesl [Zhang and Mueller 2012] provide architecture-specific mapping of nested data parallelism to the hierarchical structure of GPUs, but this mapping is embedded in their compiler implementations and not exposed to autotuning. The concept of separating specification and implementation has appeared in flat data-parallel models [Bell and Hoberock 2011; Jones and Singh 2009] and in systems that target specific domains such as image processing [Ragan-Kelley et al. 2013] and graph workloads [Prountzos et al. 2012]. In addition, recent work has explored the integration of autotuning into parallel programming systems [Chang et al. 2016; Steuwer et al. 2015]. In contrast, Surge uniquely offers the generality and ease-of-use of a nested data-parallel system and the portability of separating implementation and autotuning.

This article makes the following contributions:

—Describes a new nested data-parallel programming interface, implemented as a C++ library, that decouples high-level specification of computations from low-level implementation details using two constructs: schedules and policies.
—Introduces techniques to automatically analyze nested data-parallel specifications and generate a search space of valid low-level platform-specific code using schedules and policies. Autotuning then selects the best implementation for the execution context.
—Demonstrates Surge is capable of achieving performance on par with or better than manually optimized CUDA and OpenMP implementations across five benchmark applications.

2. PROGRAMMING INTERFACE

Surge exposes a nested data-parallel programming interface, implemented as a C++ library. Programs written using this interface can be compiled to platform-specific code using a standard C++11 compiler. Table I lists the currently supported data-parallel operators, and Listing 1 shows an example of how they may be used to express sparse matrix-vector multiplication (SpMV). A basic `sequence` type, denoting a view over

Table I. Current Data-Parallel Operators in Surge. Parameters in Square Brackets Are Optional

Operator	Description
`map(f, s1,..., sn)`	Produces sequence (`f(s1[0], ..., sn[0]), f(s1[1], ..., sn[1]), ...`)
`reduce(⊕, s, p)`	Produces the result (`p ⊕ s[0] ⊕ s[1] ⊕ ...`) for a commutative and associative operator ⊕
`reduce_by_key(⊕, s, k, p)`	Performs segmented reduction of sequence s with key sequence k. Prefix p represents the initial value of reduction.
`scan(f, s, p)`	Produces sequence `y` s.t. `y[0] = p` and `y[i] = f(y[i-1],s[i-1])` for associative operator `f`
`gather(s, idx)`	Produces sequence `y` s.t. `y[i] = s[idx[i]]`
`scatter(s, idx, d)`	Updates sequence `d` s.t. `d[idx[i]] = s[i]`
`range(s, e, [stride])`	Produces sequence with values ranging from `s` to `e` with stride `stride`
`replicate(v, len)`	Synthesizes sequence `s` of length `len` s.t. `s[i] = v` for all `i`
`zip(s1,...,sn)`	Produces sequence `x` s.t. `x[0]`=⟨`s1[0], ..., sn[0]`⟩, `x[1]`=⟨`s1[1], ..., sn[1]`⟩, ...
`split(s, l)`	Produces nested sequence `x` from `s` s.t. each sub-sequence of `x` is a tile of `s` of size `l`
`cyclic(s, l)`	Produces nested sequence `x` from `s` s.t. `x` is the transpose of nested sequence created by `split(s, l)`
`join(s1, ..., sn)`	Produces sequence (⟨$x_1, \ldots, x_n$⟩$_i$) s.t. $x_1 \in s_1$, $x_2 \in s_2, \ldots$ & $i \in [0, \prod_{i=1}^{n} len(s_i))$
`striding(s, stride)`	Produces strided sequence from `s` of stride `stride`
`reverse(s)`	Produces the reversed sequence of `s`
`nest(s, i)`	Produces a nested sequence from `s` with subsequence offsets `i`

```
1  // Create nested sequences s_matrix and s_indices
2  auto s_matrix = nest(s_nonzeros, s_row_offsets);
3  auto s_indices = nest(s_column_indices, s_row_offsets);
4  using row_t = decltype(s_matrix[0]);
5  using index_row_t = decltype(s_indices[0]);
6
7  auto spmv =
8    // Apply dot product across all rows of matrix
9    map([=] (row_t row, index_row_t indices) {
10         auto mul = [](value_t x, value_t y) { return x*y; };
11         auto plus = [](value_t x, value_t y) { return x+y; };
12         // Gather elements from vector s_vector
13         auto z = gather(s_vector, indices);
14         // Element-wise multiplication of vector with row
15         auto vector_mul = map(mul, row, z);
16         // Sum up elements to obtain dot product
17         return reduce(plus, vector_mul, value_t(0));
18       },
19       s_matrix, s_indices);
20
21 // Realize SpMV computation
22 execute(spmv, s_result);
```

Listing 1. Surge code for Sparse Matrix-Vector Multiplication (SpMV). See the Appendix for the full code listing and Section 5.2 for a more detailed description.

contiguous one-dimensional data, is also provided. More complex types of sequences can be built up using operators such as `nest`, as described in Table I. In Listing 1, for example, `s_matrix` and `s_indices` are nested sequences (represented internally using the compressed sparse row (CSR) matrix format) constructed from the flat one-dimensional sequences `s_nonzeros` and `s_column_indices`, respectively, and the same row offset sequence `s_row_offsets`; each element of `s_matrix` and `s_indices` thus represents a single row of nonzeros and corresponding column indices of the original matrix.

For each computation in the program, Surge builds an *expression sequence* to capture the nesting structure of its data-parallel operators. Each element of an expression sequence E is a single data-parallel operator, and, given two elements e and f in E, f follows e in the sequence (represented as $e \triangleright f$) only if the operator corresponding to f is nested within that of e in the computation. In the SpMV example (Listing 1), the outermost `map` (line 9) iterates over matrix rows, and the `reduce` operator on line 17 operates on these rows and is nested within the `map`. The corresponding expression sequence is therefore `map`$\triangleright$`reduce`. Note that the `gather` and `map` on lines 13 and 15, respectively, are not part of the nesting structure; instead they are arguments to `reduce` and are fused into it as explained in Section 4.

An expression sequence is an abstract entity and can be realized in hardware in multiple ways. For example, on CUDA, two different implementations of the SpMV expression sequence can be obtained by either assigning each iteration of the outermost `map` to a thread or to a logical CUDA warp (power-of-two contiguous group of threads that are at most the physical warp size); the former corresponds to the CSR-Scalar implementation and the latter to CSR-Vector, as described in Bell and Garland [2009]. To bind an expression sequence to a concrete hardware implementation, Surge introduces two new constructs in the interface: schedules and policies. A schedule, when associated with a data-parallel operator, characterizes the dependence structure of that operator, constraining the ways in which it may be implemented; these constraints can then be systematically relaxed to obtain platform-independent implementation strategies. Policies, on the other hand, provide fine-grained control over low-level, platform-specific implementation details by encapsulating the parameters that drive code generation. By exposing schedules and policies in the programming interface, as opposed to embedding them deep in the code generation infrastructure, both autotuners and expert programmers are able to easily experiment with multiple implementations for a computation. We describe schedules and policies in more detail in Section 3.

For performance-portable code generation, we believe that decoupling computations from their implementations alone is not enough: It is equally important to be able to reason about implementation strategies for computations in a purely platform-independent manner; targeting a new platform then reduces to the problem of finding ways to customize these strategies for that platform. Surge achieves this by keeping the concepts of schedule and policy separate. In contrast, a system that generates platform-specific implementations directly from high-level specifications (regardless of whether they are decoupled or not) must re-implement its entire code generation infrastructure for each new platform.

Invoking the `execute` function initiates the process of binding the expression sequence to a concrete implementation. It has the following form:

```
execute(expr, [destination, platform, schedule, policy])
```

Here `expr` is the nested data-parallel computation, and the optional `destination` argument specifies where to copy the results of the computation. The `platform` argument is used to specify the target hardware platform. Surge currently supports two platforms: GPUs and x86 CPUs through CUDA C++ and OpenMP, respectively. If this argument is left unspecified, it defaults to CUDA C++. Schedules and policies may be specified through the `schedule` and `policy` parameters, respectively; in this article, they are inferred automatically via autotuning (as described in Section 3). The values of the parameters `platform, schedule,` and `policy` determine a unique implementation for the computation in `expr`. Once specified, `expr` is automatically compiled to either CUDA C++ or OpenMP code using static meta-programming (described in Section 4) and a standard C++ compiler.

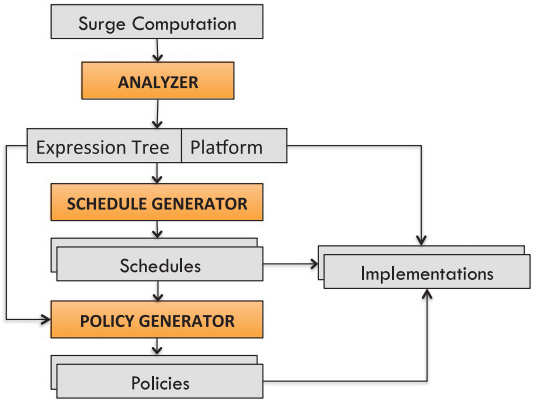


Fig. 1. Overview of the Surge code generator.

Table II. List of Surge Schedules

Schedule	Description
`independent`	Permits the use of multiple execution resources working in parallel
`cooperative`	Permits multiple resources, but they may additionally coordinate with each other
`sequential`	Permits the use of a single thread

3. CODE GENERATION AND AUTOTUNING

The Surge code generator analyzes nested data-parallel computations in the program and for each one, systematically enumerates the set of semantically valid schedules and policies. The code generator is implemented as a set of Python modules. Figure 1 provides an overview of the code generation process.

3.1. Computation Analysis

The job of the analyzer is to extract the expression sequence and platform information for each computation in the program. We avoid using a full-fledged C++ parser for this and instead rely on a lightweight *introspection* mechanism. The analyzer recompiles the input program with the macro `INTROSPECTION_MODE` defined; this instructs Surge to pretty-print the expression sequence and platform information of the computation instead of evaluating it. The resulting program is run and its output is parsed by the analyzer. As a concrete example, see Listing 8 in the Appendix; here, the `INTROSPECTION_MODE` macro is automatically defined in the `surge_config.h` file (line 15) and is included before the `execute.h` header (line 19), thus enabling it to change the behavior of the `execute` function. Since expression sequence elements are encoded as templated types (as described in Section 4), static meta-programming is used to recursively traverse the expression sequence and print out its information. This is similar to the approach in VexCL for implementing its `symbolic` type [Demidov et al. 2016].

3.2. Schedule Enumeration

Schedules are defined in terms of *execution resources*, which are platform-specific units capable of carrying out a data-parallel operation. Examples of execution resources include threads, CUDA warps, OpenMP thread-pools, and so on. Surge currently supports three schedules: `independent`, `cooperative`, and `sequential` (Table II). Schedules are nested to correspond with the nesting structure of the associated expression sequence. For the SpMV code, an example valid schedule is `independent▷cooperative`: The

Table III. Schedule Lookup Table for Surge Operators

Operator(s)	Strongest Schedule
`map, gather, scatter, range, replicate, zip, join, striding, reverse`	`independent`
`split, cyclic, nest`	`independent▷independent`
`reduce, reduce_by_key, scan`	`cooperative`

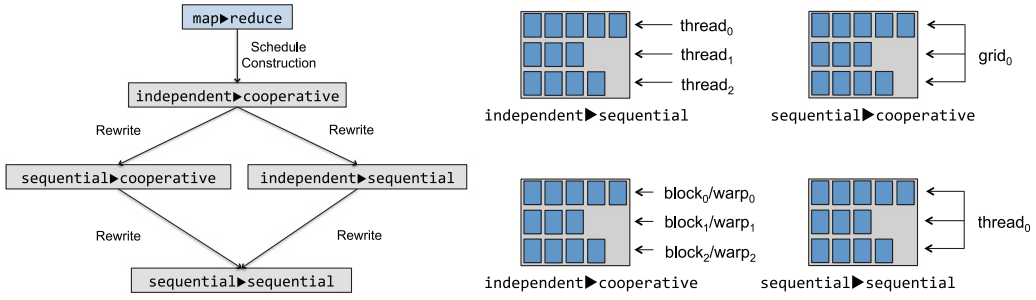


Fig. 2. (a) SpMV schedule construction and rewriting and (b) how various SpMV schedules may be implemented in CUDA; in this example, the input matrix (gray boxes) has 12 nonzeros (blue boxes) and 3 rows.

outermost `map` can process its elements independently, whereas the inner `reduce` (vector dot product) requires a cooperation stage among threads if implemented in parallel.

Schedule enumeration refers to the process of discovering the set of valid schedules for a given expression sequence and platform. It consists of two phases: (1) schedule construction and rewriting and (2) platform-specific pruning. Before describing schedule enumeration, we first introduce the *schedule rewrite rules*, which make it possible to transform one schedule to another in a well-defined manner:

$$\texttt{independent} \rightarrow \texttt{sequential}$$
$$\texttt{cooperative} \rightarrow \texttt{sequential}.$$

We define the *strength* binary relation over the set of schedules as follows: A schedule s_1 is said to be stronger than schedule s_2 iff s_2 can be obtained from s_1 by following the rewrite rules in Surge. Operators implemented using a schedule s can always be implemented using any schedule weaker than s. For example, a `map` operator, implemented using the `independent` schedule, can always also be implemented using the weaker `sequential` schedule. On nested schedules, rewrite rules are applied one at a time on individual elements.

Schedule Construction and Rewriting. The first step of schedule enumeration is inferring the *strongest* nested schedule for the given expression sequence—this is the schedule construction phase. Elements of the input expression sequence are traversed in order, and a *schedule lookup table* (shown in Table III) is consulted to obtain the corresponding element in the strongest schedule's sequence.[1] The strongest obtained schedule is then systematically rewritten to obtain the set of all its weaker schedules. Figure 2(a) depicts schedule construction and rewriting for the SpMV example from Listing 1, and Figure 2(b) visualizes how the generated schedules may be implemented on the CUDA platform.

[1]Note: nonassociativity of floating-point operations is not considered during this mapping.

Table IV. List of Tunable Parameters

Parameter(s)	Type	Platform
`block_size_x, block_size_y, logical_warp_size`	Global	CUDA
`grain_size, block_reduce_algo, block_scan_algo, block_scan_grain_size`	Local	CUDA
`num_threads, enable_nesting`	Global	OpenMP
`omp_schedule, chunk_size`	Local	OpenMP
`execution_resource, enable_unroll`	Local	CUDA/OpenMP

Platform-Specific Pruning. While generated schedules are guaranteed to be semantically valid, they may not always be directly *implementable* on the given platform. For example, since CUDA only supports two levels of parallelism on a single GPU (at the thread block or warp level and at the thread level), any schedule after nesting level 2 must be `sequential`. Surge thus defines a set of *schedule constraints* for each platform, and any schedules that violate one or more of these constraints are removed from further consideration.

3.3. Policy Enumeration

A policy for an expression sequence E consists of a set of *global* and *local* parameters; the former affect the implementation of the entire computation, while the latter that of the associated element of E. An example of a global parameter is `warp_size` in CUDA, which specifies the number of threads in a logical CUDA warp, while that of a local parameter is `omp_schedule`, which specifies the OpenMP loop schedule to use. Table IV lists the parameters currently supported by Surge.

As shown in Figure 1, the final stage of the code generation process is policy enumeration. Let S be the set of valid schedules produced for expression sequence E and platform B. The policy enumerator, for each $s \in S$, generates a set of platform-specific *optimization parameter values* that dictates how the tuple $\langle E, B, s\rangle$ is implemented in hardware. We now describe the two phases of policy enumeration: optimization parameter inference and search space generation.

Optimization Parameter Inference. The set of valid optimization parameters T is given by:

$$T = \bigcup_{s \in S} \big(G_{\langle E,B,s\rangle} \cup \texttt{Get-Parameters}(E, B, s)\big),$$

where G is the set of global parameters, and `Get-Parameters` (shown in Algorithm 1) is a function that returns the set of valid local parameters for each $\langle E, B, s\rangle$ tuple. The inferred parameters for the schedules in the SpMV example (see Figure 2(a)) are shown in Table V.

ALGORITHM 1: Parameter Inference

```
1:  function GET-PARAMETERS(E, B, s)
2:      #E: Expression sequence of form O1▷O2▷···▷On▷φ
3:      #s: Schedule sequence of form s1▷s2▷···▷sn▷φ
4:      #B: Platform
5:      if E = φ then return ()
6:      else
7:          t ← parameter-set[O1, s1, B]
8:          E' ← O2▷O3▷···▷On▷φ
9:          s' ← s2▷s3▷···▷sn▷φ
10:         return t ∪ GET-PARAMETERS(E', B, s')
```

Table V. Inferred Parameters for each SpMV Schedule. The Subscripts Denote Nesting Depths

Schedule	Inferred Parameters (CUDA)	Inferred Parameters (OpenMP)
independent▷cooperative	block_size_x, block_size_y, logical_warp_size, grain_size$_1$, block_reduce_algo$_2$	num_threads, enable_nesting, omp_schedule$_1$, chunk_size$_1$, omp_schedule$_2$, chunk_size$_2$
independent▷sequential	block_size_x, block_size_y, logical_warp_size, grain_size$_1$, enable_unroll$_2$	num_threads, enable_nesting, omp_schedule$_1$, chunk_size$_1$, enable_unroll$_2$
sequential▷cooperative	block_size_x, block_size_y, logical_warp_size, block_reduce_algo$_2$	num_threads, enable_nesting, omp_schedule$_2$, chunk_size$_2$, enable_unroll$_1$
sequential▷sequential	block_size_x, block_size_y, logical_warp_size, enable_unroll$_1$, enable_unroll$_2$	enable_unroll$_1$, num_threads, enable_nesting, enable_unroll$_2$

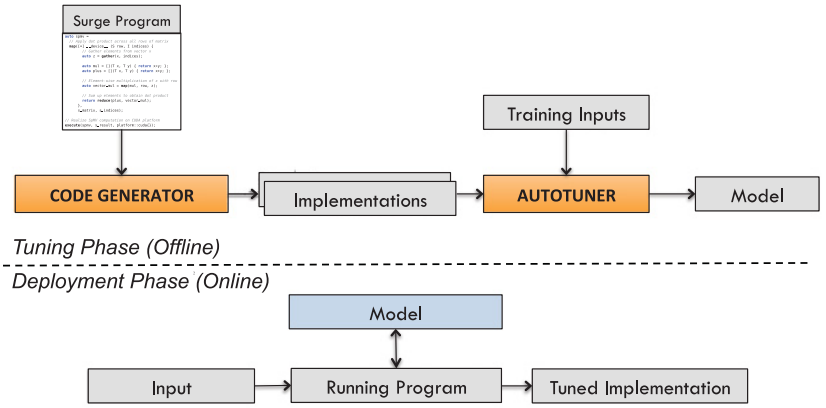


Fig. 3. Overview of the Surge framework and its interaction with the autotuner.

Search Space Generation. Each parameter $t \in T$ can take on a set of values. If r is a function that takes a parameter as input and outputs the list of its valid values, then the search space is $\prod_{t \in T} r(t)$, where $\prod$ denotes a Cartesian product. However, since not all points in this space may be valid on the given platform, search space generation is followed by a pruning phase that discards points that violate platform-specific constraints. For example, while the maximum dimension size of a CUDA thread block along the x- and y-axes is 1,024 each, the total number of threads per block (product of x- and y-axis dimensions) is also restricted to 1,024 on current GPUs such as the NVIDIA Tesla K20c.

3.4. Autotuning

The code generation approach from the previous section produces a search space of functionally equivalent implementations (called *code variants*) for each computation. To automate the selection of which version is most appropriate for a given execution context, we rely on autotuning. The Surge framework and its interaction with the autotuner is depicted in Figure 3. While autotuning is becoming widely used as part of programming systems [Chang et al. 2016; Steuwer et al. 2015], a few search

frameworks can be used as stand-alone tools [Tiwari et al. 2009; Hartono et al. 2009; Muralidharan et al. 2014; Ansel et al. 2014]. In this article, we use Nitro for code variant selection [Muralidharan et al. 2014]. Nitro facilitates code variant selection that fully adapts to execution context; it adapts to the architecture as well as the input dataset. Since nested data parallelism specifically operates on multi-dimensional (nested) data, having the ability to adapt to input data characteristics is valuable.

The programmer provides a set of training inputs that are representative of input datasets; synthetic inputs can be used for regular codes whose performance is only dependent on problem size, but representative inputs are needed for irregular nested data-parallel computations such as SpMV. For each training input, the autotuner builds a *feature vector* that encodes the input's characteristics and measures the performance of each code variant in the search space to find the best-performing one. Surge infers three features automatically by analyzing the structure of the first input sequence used by the computation: (1) length of the input sequence, (2) aspect ratio for uniformly nested sequences, and (3) average row length for irregularly nested sequences (obtained through the `nest` operator, for example). Nitro builds a Support Vector Machine (SVM) [Vapnik 1998] *model* in this offline training phase (Figure 3 (top)) and stores it as a text file on disk. The model maps from features of the input dataset to code variants and is automatically loaded and consulted at application runtime to select an optimal variant for the given input (Figure 3 (bottom)).

4. TRANSLATION TO TARGET-SPECIFIC CODE

The Surge programming interface is a domain-specific embedded language [Hudak 1996] with C++ as the host. The primary entities in the programming interface, namely operators, schedules, policies, and platforms, are all implemented as types to enable static meta-programming. In particular, Surge overloads operators to act as type constructors, as in the expression template idiom [Veldhuizen 1995], to construct the expression sequence at compile time. This enables the hardware realization of operators to be deferred until an appropriate implementation context (platform, schedule, and policy) is available. As a concrete example, consider `map`: It is recorded as the type `transformed_sequence<F, S1, S2, ...>`, where `F` is the function being applied to every element of sequences S_i. Since a `map`'s iterations can be executed independently, `transformed_sequence` correspondingly provides the subscript operator to realize each individual iteration independently. Thus, in Listing 1, the `spmv` variable on line 7 is a `transformed_sequence`, and `spmv[0]` calls the lambda function defined on line 9 with arguments `(s_matrix[0], s_indices[0])`; this returns an object of type `reduced_sequence`, corresponding to the `reduce` operator on line 17.

With the expression sequence constructed, a set of *nested computation kernels*, defined for each platform, is used to realize computations. Each kernel implements a set of $\langle E, B, S, P\rangle$ tuples, where E is the expression sequence, B is the platform, S is the schedule, and P is the policy. Representing schedules and policies as separate types enables us to utilize the C++ *substitution failure is not an error (SFINAE)* idiom and function overloading to define both generic and highly specialized computation kernels conveniently. For example, Listing 2 shows a simple kernel that realizes any valid operator bound with the `independent` schedule on CUDA. In contrast, the type signature of a kernel specialized for the tuple ⟨`reduce_by_key`▷`_`, `CUDA`, `cooperative`▷`independent`, `_`⟩ is as follows:

```
template<typename S, typename D, typename Policy>
__global__ void nested_kernel(S src, D dest,
                cooperative<independent<>>, Policy,
                mpl::enable_if_t<mpl::sequence_traits::
                  is_seg_reduced<S>::value>* = 0);
```

```
1  // Implements tuple <_, CUDA, independent, _>
2  template<typename S, typename D, typename Policy>
3  __global__ void nested_kernel(S src, D dest,
4                  independent<>, Policy) {
5    using iteration_type =
6      typename S::template iteration_type<
7      D, platform::cuda, res::cuda::thread,
8      mpl::global_policy_t<Policy>,
9      mpl::sub_policy_t<Policy, 1>>;
10
11   const int idx = blockIdx.x*blockDim.x+threadIdx.x;
12   const int grid_size = gridDim.x*blockDim.x;
13
14   iteration_type iterator(src, dest);
15   for(int i = idx; i < src.size(); i += grid_size)
16     iterator[i];
17   iterator.finalize();
18 }
```

Listing 2. Sample CUDA nested computation kernel for the `independent` schedule.

and for the tuple ⟨_, CUDA, independent▷cooperative, ⟨execution_resource=cuda_warp, ···⟩⟩ is as follows:

```
template<typename S, typename D, typename Policy>
__global__ void nested_kernel(S src, D dest,
                independent<cooperative<>>, Policy,
                mpl::enable_if_t<std::is_same<typename
                  policy::sub_policy::execution_resource,
                  res::cuda::warp>::value>* = 0)
```

Targeting New Architectures. Implementing support for a new architecture involves defining a new platform type, and corresponding tunable parameters and nested computation kernels. As described above, the two-layered schedule+policy approach provides a great amount of flexibility while implementing new computation kernels: Programmers can start with fairly generic kernels and then specialize incrementally. This separation also reduces the effort required to add automatic code generation support, as every phase of the code generator need not be re-implemented; instead, only the platform-specific schedule pruning and policy enumeration phases need to be implemented for the new platform, as described in Section 3.

Note on Operator Fusion. Deferred realization permits operators to be fused together: Consider the case when an operator O_p is an argument to operator O_c; for example, in Listing 1, `gather` (on line 13) is an argument to `map` (line 15), which in turn is an argument to `reduce` (line 17). If O_p can be realized using independent iterations, then Surge fuses each iteration of O_p (producer) with that of O_c (consumer) and thus eliminates the need for temporary storage required to realize O_p.

5. BENCHMARKS

We express five benchmark applications in Surge and evaluate their performance on both multi-core CPUs and GPUs. To better model a range of real-world applications, the benchmarks are of varying complexity and are drawn from diverse domains such as linear algebra, machine learning, and physical simulation. Table VI lists our benchmarks, together with the nested operators used and details about the reference implementations. The remainder of this section describes each benchmark in detail.

5.1. Reduction and Scan

For our first set of benchmarks, we implement parallel reduction and parallel prefix scan (scan for short) in Surge. Reduction and scan are fundamental parallel computing

Table VI. List of Benchmarks with Description, Their Core Computation(s) and Details about Reference Implementations

Benchmark	Description	Core Computation(s)	Reference Implementation
`Reduction`	Parallel reduction	`map▷reduce`	Thrust 1.8.2 [Bell and Hoberock 2011]
`Scan`	Parallel prefix scan	`map▷scan`, `map▷reduce`	Thrust 1.8.2 [Bell and Hoberock 2011]
`SpMV`	Sparse matrix-vector multiplication	`map▷reduce`	GPU: CUSP 0.5.1 [Dalton et al. 2010], CPU: MKL 11.2 [Intel 2016]
`K-Means`	K-Means clustering using LLoyd's algorithm [Lloyd 1982]	`reduce_by_key▷map`, `map▷reduce`	Catanzaro [Catanzaro 2014]
`CoMD`	Co-design molecular dynamics proxy application	`map▷reduce`, `map`	ExMatEx CoMD 1.1 [ExMatEx 2015; Sakharnykh 2013]

primitives that are widely used as building blocks for more complex algorithms [Merrill 2011]. Given a sequence of input elements $x_0, x_1, x_2, \ldots, x_N$, a prefix element p, and a binary operator $\oplus$, the output of a reduction is the scalar value $x = p \oplus x_0 \oplus x_1 \oplus \cdots \oplus x_N$ while that of prefix scan is the sequence $y_0, y_1, y_2, \ldots, y_N$, where $y_0 = p$ and each $y_i = y_{i-1} \oplus x_{i-1}$.

In the Surge implementation (Listings 3 and 4), the input sequence is first split into evenly sized tiles that are reduced or scanned in parallel to yield a set of partial results (or partials). These are processed to obtain the final result of the reduction or scan. For the computation of partials, the in-built `reduce` and `scan` operators are instantiated within a `map`, yielding a nested data-parallel algorithm.

5.2. Sparse Matrix-Vector Multiplication

SpMV is a critical operation that is used in many iterative methods for solving large-scale linear systems. For this benchmark, we implement the SpMV computation in Surge, as shown in Listing 1. The sparse matrix and column indices (`s_matrix` and `s_indices` on line 19) are represented as nested sequences, which are internally stored in a CSR analogue. The outermost `map` (line 9) processes each row of the matrix. Inside the body of the lambda that processes a single row, the correct elements of the vector are first gathered (line 13), multiplied on an element-wise basis with the current matrix row (line 15), and, finally, summed up to yield the dot product of that row (line 17). Note that the `gather` and `map` on lines 13 and 15 are automatically fused into the `reduce` on line 17, eliminating temporaries (see Section 4 for a description of operator fusion).

5.3. *k*-Means Clustering

k-Means clustering is an important algorithm commonly used in fields such as computer vision and signal processing. The problem is defined as follows: given a set of N data points in D-dimensional space R^D, and an integer k, determine a set of k points in R^D, called *centroids*, to minimize the mean-squared distance from each data point to its nearest centroid. We implement a popular heuristic for k-means clustering called Lloyd's algorithm [Lloyd 1982]. Additionally, we use the strategy outlined by Catanzaro [Catanzaro 2014] and rewrite the distance computation $||x - y||^2$ as $x \cdot x + y \cdot y - 2 \cdot x \cdot y$, where x denotes a point and y a centroid. This refactorization lifts the $x \cdot x$ computation out of the main k-means loop and enables the use of vendor-optimized GEMM library calls to efficiently compute $x \cdot y$.

Given an initial set of k means, Lloyd's algorithm proceeds by alternating between two steps: (1) relabeling, which is assigning each point to the cluster with the nearest

```
auto plus =
  [] (value_t a, value_t b) { return a + b; };

// Split original flat sequence (s) into C tiles
auto s_tiled = split(s, tile_size);

using row_t = decltype(s_tiled[0]);
auto row_reduce =
  [=] (row_t row) { return reduce(plus, row, value_t(0)); };

// Compute per-tile reductions
execute(map(row_reduce, s_tiled), s_partials);

// Reduce partials into s_result[0]
auto s_partials_tiled = split(s_partials, C);
execute(map(row_reduce, s_partials_tiled), s_result);
```

Listing 3. Surge code for parallel reduction.

```
auto plus =
  [] (value_t a, value_t b) { return a + b; };

// Split original flat sequence (s) into C tiles
auto s_tiled = split(s, tile_size);
using row_t = decltype(s_tiled[0]);

// Compute per-tile reductions
execute(
  map([=] (row_t tile) {
        return reduce(plus, tile, value_t(0));
      }, s_tiled),
  s_partial_reductions);

// Prefix sum over partial reductions
auto s_partials_tiled = split(s_partial_reductions, C)
execute(
  map([=] (row_t tile) {
        return scan(plus, tile, value_t(0));
      }, s_partials_tiled),
  s_partial_scans);

// Compute full prefix sum by seeding from reduction
execute(
  map([=] (row_t tile, T prefix) {
        return scan(plus, tile, prefix);
      }, s_tiled, s_partial_scans[0]),
  s_result);
```

Listing. 4. Surge code for parallel prefix scan.

centroid, where the distance between points is the Euclidean distance, and (2) centroid recalculation, which is calculating the new cluster centroids as the mean of the values of points in the new clusters. While the Surge *k*-means implementation consists of five computations, we focus on the more expensive centroid recalculation step, the code for which is shown in Listing 5. Here the centroids and data points are stored as tiled sequences (with tile size *D*) and are obtained by applying a `split` on flat 1D sequences stored in row-major format (`s_centroids_flat` and `s_points_flat`). Each element of `s_labels`, say, `s_labels[i]`, initially contains the label (index) of the closest centroid for data point `i`. We sort `s_labels` to bring all labels of the same centroid together and store the corresponding point indices in `s_indices` (which initially holds `range(0,`

```
1  // Use a tiled sequence for centroids and points.
2  auto s_centroids = split(s_centroids_flat, d);
3  auto s_points = split(s_data_flat, d);
4
5  // Bring all labels with the same value together
6  thrust::sort_by_key(s_labels.begin(), s_labels.end(), s_indices.begin());
7
8  auto s_points_x = gather(s_points, s_indices);
9  auto s_prefix = replicate(0, d);
10
11 using point_t = decltype(s_points_x[0]);
12
13 auto plus =
14    [] (value_t a, value_t b) { return a + b; };
15
16 execute(
17      reduce_by_key([=] (point_t x, point_t y) {
18                      return map(plus, x, y);
19                    }, s_points_x, s_labels, s_prefix),
20      s_centroids);
```

Listing 5. Surge code for k-means centroid recalculation.

$N-1$)). The sum of the points belonging to each centroid can now be obtained through a segmented reduction of `s_points` (permuted through `s_indices`) with key `s_labels`. Since each point is itself in D-dimensional space, we use `reduce_by_key`▷`map` (line 17) to accomplish this. A simple scaling step (not shown in the Listing) then divides the resulting points by their correct counts to obtain the new set of centroids.

5.4. Co-Design Molecular Dynamics Proxy

Co-design Molecular Dynamics Proxy (CoMD) is a molecular dynamics proxy application that is part of the ExMatEx project [ExMatEx 2015]. The workloads seen in the reference CoMD application are representative of those in classical molecular dynamics applications, which is to identify all pairs of atoms under a radius cutoff and compute the force between these pairs. While the reference implementation supports methods of computing Lennard-Jones and Embedded Atom Method (EAM) potentials, we only consider the EAM potential method in this article.

For this benchmark, we express two algorithms for the EAM potential method for computing inter-atom forces in Surge. One computes the forces directly (Listing 6), while the other performs a domain-specific redistribution of atoms to expose more parallelism before computing the forces (Listing 7). The direct method splits the input atom space into evenly sized tiles and computes partial energy values for each tile. The partials are then reduced to obtain the final energy value. In the redistributed version (Listing 7), the `map` on line 8 applies the `eam_force_functor`, which updates a given atom's force, energy, and position, to each atom. Note that both algorithms are expressed in a platform-neutral way and are targetable on both hardware platforms. To enable selection between these two algorithms, we specify them as algorithmic variants using Nitro. We thus obtain a two-level selection process where the algorithm is first selected, followed by the implementation of that algorithm for the target platform.

6. EVALUATION

In this section, we demonstrate performance and productivity results for the five benchmarks described in Section 5. For each benchmark, both CPU and GPU implementations are automatically generated, and the performances of the best ones for each platform (found through autotuning) are compared against handwritten reference implementations for that platform.

```
auto s_space = split(s_boxes, tile_size);
auto s_count_range = range(0, s_space.size());

using tile_t = decltype(s_space[0]);

// Compute partial reductions into s_result
execute(
  map([=] (tile_t tile) {
    real_t etot = 0.;

    // Loop over neighboring atoms,
    // update force and compute energy.
    ...

    return etot;
  }, s_space, s_count_range),
  s_result);

// Reduce partials
real_t etot = thrust::reduce(s_result.begin(), s_result.end());
```

Listing 6. Surge code for CoMD inter-atom force calculation (direct version).

```
auto s_domain = range(0, atoms_list.n);

// eam_force_functor updates the
// given atom's force, energy and position.
eam_force_functor f(sim, atoms_list);

// In-place execute
execute(map(f, s_domain));
```

Listing 7. Surge code for CoMD inter-atom force calculation (redistributed version).

6.1. Methodology and Hardware Platforms

Our evaluation was run on two hardware platforms: (1) a dual-socket, 32-core Intel Xeon E5-2698 v3 CPU (Haswell) running at 2.30GHz and (2) an NVIDIA Tesla K20c GPU (Kepler generation). The NVIDIA CUDA compiler (NVCC) 8.0RC was used, and g++-4.8.2 with OpenMP 3.1 was used as the host (CPU) compiler. The `-O3` flag was specified. For the Intel Math Kernel Library (MKL) results collection, the Intel compiler v15.0 was used with the `KMP_AFFINITY` environment variable set to `granularity=fine,scatter`. All implementations were run for 100 timing iterations to collect consistent results. Unless otherwise specified, double precision floating-point numbers were used in our evaluation.

Once the benchmarks were specified, Surge automatically generated valid implementations for the desired platform, tuned them, and produced the SVM models. Table VII shows the tuning information for each benchmark, including the features inferred automatically by Surge (as explained in Section 3.4), number of training and testing inputs, and size of the search space (number of distinct implementations generated by Surge). When Nitro builds its model in the offline training phase, it automatically finds the best variant corresponding to each training input using exhaustive search; the maximum number of such unique variants across all computations in a benchmark is listed in the last two columns of Table VII. We observe that although the initial search space is fairly large, the set of variants for a computation that perform well on a given platform is relatively small.

Training and Testing Inputs. As mentioned in Section 3.4, the training inputs for Nitro, due to their domain-specific nature, must be provided by the programmer. For

Table VII. Features Used, Number of Training and Test Inputs, Size of Search Space, and Number of Variants for Each Benchmark

Benchmark	Inferred Features	#Inputs		Search Space Size		#Variants	
		#Training	#Testing	GPU	CPU	GPU	CPU
`Reduction`	`#tiles, aspect_ratio`	GPU:8, CPU:6	8	42	36	4	3
`Scan`	`#tiles, aspect_ratio`	GPU:7, CPU:5	8	90	72	4	3
`SpMV`	`#rows, avg_rowlen`	GPU:10, CPU:6	13	42	36	5	4
`K-Means`	`#elements, #tiles, aspect_ratio`	GPU:5, CPU:7	7	40	12	3	5
`CoMD`	`#elements, #tiles, aspect_ratio`	GPU:7, CPU:7	7	48	60	4	3

reduction, scan, *k*-means, and CoMD, we generated synthetic inputs for both training and testing; for SpMV, we used sparse matrices from the UFL Sparse Matrix Collection [Davis 2011]. To obtain representative training sets, we start with a large pool of inputs for each benchmark and use Nitro's active learning heuristic [Muralidharan et al. 2014] to automatically prune it down and obtain the final training set. The test sets are mutually distinct from the training set. The third column of Table VII shows the number of training and test inputs for each benchmark.

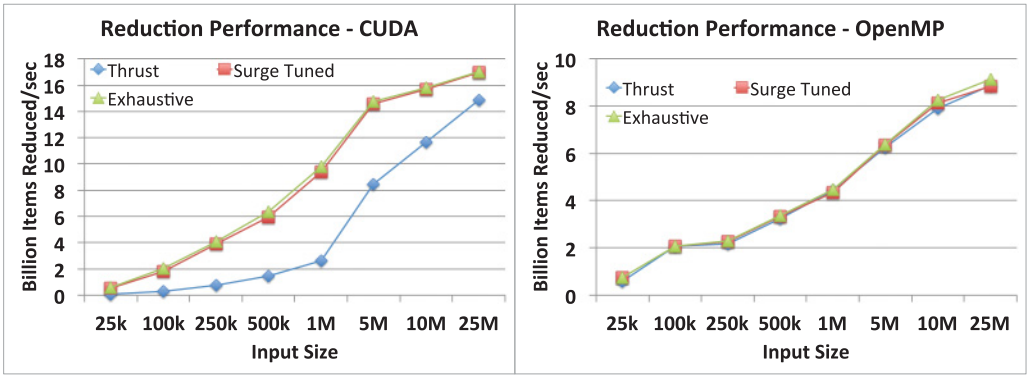
6.2. Performance Results

Figures 4 and 5 show performance results for our benchmarks on both hardware platforms. In each graph, points on the x-axis represent different inputs from the test set, while the y-axis shows performance in terms of throughput. For each benchmark, we show the performance achieved by the reference and tuned implementations. Note that the performance data shown for the tuned implementations include feature evaluation and SVM model query time. We also include a comparison with the performance achievable if the best implementation among all the generated ones were found via exhaustive search for each test input (lines and bars labeled "Exhaustive"). The average speedups (over the test set) achieved by the tuned Surge implementation over reference implementations are listed in the second and third columns of Table VIII.

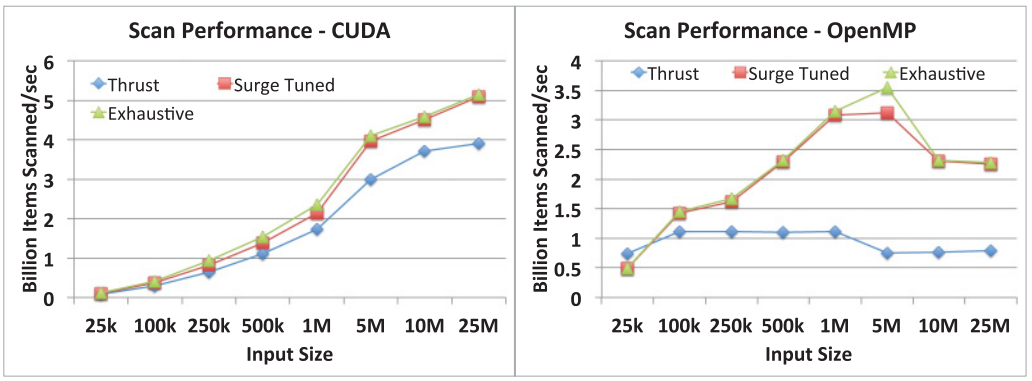
Reduction and Scan. Figures 4(a) and 4(b) show the performance of reduction and scan. The tuned version either matches or significantly outperforms the reference implementations on both platforms for all test inputs. The performance is especially good on the GPU for small input sizes, where a CUDA warp-based reduction or scan is automatically selected. For scan on the CPU, it appears that Thrust internally uses a sequential algorithm, resulting in considerably lower throughput compared to the parallelized Surge version.

SpMV. Figure 4(c) shows the performance of the Surge SpMV code (shown in Listing 1). We compare against the CUSP CSR Vector and MKL CSR implementations on the GPU and CPU, respectively. The ability to vary logical warp sizes proves to be crucial to obtaining good performance on the GPU, as matrices with smaller average row lengths perform best with smaller logical warp sizes. On matrices with relatively large average row lengths (for example, `rail4284`), the preferred execution resource on CUDA seems to be blocks, as opposed to warps. On the CPU, the tuned version performs between 80.5% and 128.8% of highly tuned MKL code.

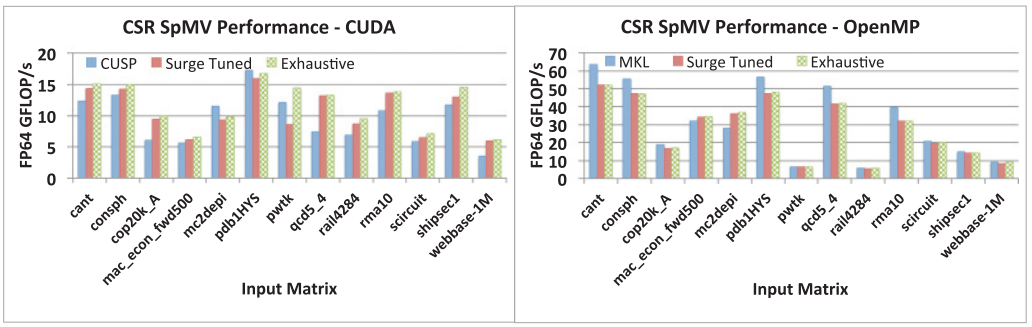
k-Means. For this benchmark, the dimension of each vector is set to 32 and the number of clusters to 10. The algorithm is run for 100 iterations. The Surge version of *k*-means uses a `reduce_by_key`▷`map` operator at the core. On CUDA, the number and size (along both the x and y dimensions) of the CUDA blocks turn out to



(a)



(b)

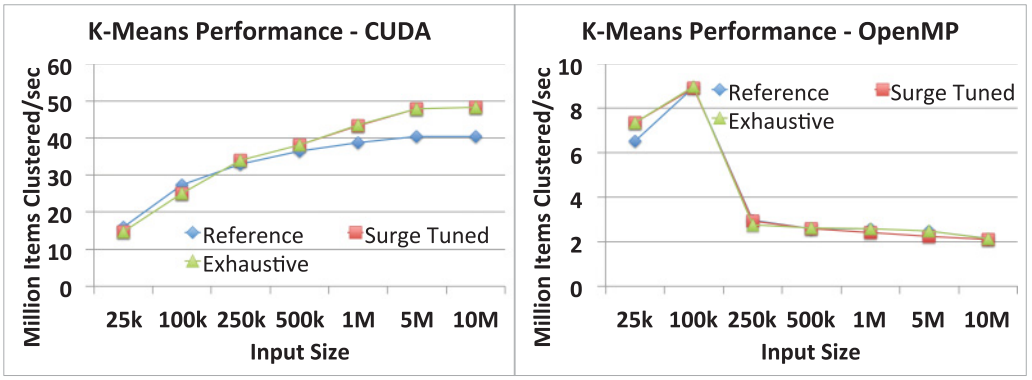


(c)

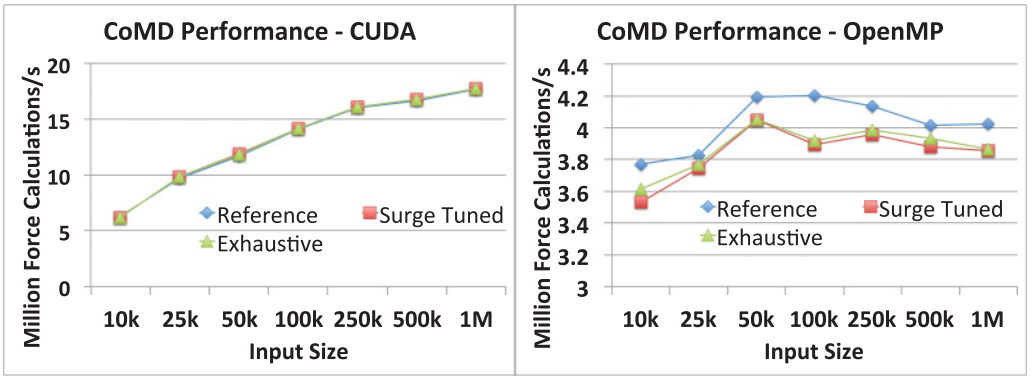
Fig. 4. Reduction, Scan, and CSR SpMV Performance on CUDA and OpenMP.

be the most important parameters, and tuning them enables us to beat the reference implementation for larger input sizes. Figure 5(a) shows the results for this benchmark.

CoMD. Figure 5(b) shows the performance numbers for CoMD. On the CPU, we see that the direct approach, which uses `map▷reduce`, works best, while the version that performs redistribution does well on the GPU. On both platforms, the version selected by Nitro performs on par with reference implementations.



(a)



(b)

Fig. 5. *k*-Means and CoMD Performance on CUDA and OpenMP.

Tuning and Overheads. Comparing the performance of implementations tuned by Surge with that of ones found via exhaustive search, we notice that the automatically constructed SVM models predict the right implementation for the given test inputs in almost all cases. This implies that the input features added by Surge are highly effective at predicting good implementations. Also, since the inferred features can be computed in constant time, and the number of variants is relatively small for all benchmarks, we observed that the overhead of feature evaluation and SVM model query was negligible (order of a few microseconds). Since applications may be drawn from various domains, we do not claim that the inferred features will always be sufficient, or that the feature evaluation and SVM model query times will always be this low; instead, we believe that the inferred features, and the integration with Nitro in general, provide a good starting point for tuning the implementations generated by Surge.

Summary. Overall, the tuned implementations generated by Surge achieve excellent performance across the board and often beat the performance of the reference implementations. On the GPU, the ability to vary the execution resource, logical warp sizes, and the number and size of blocks has the most effect on performance. On the CPU, tuning has a less pronounced effect. This is partly because most of the benchmarks operate on uniformly tiled sequences and hence perform well with the default OpenMP schedule. The notable exception is SpMV, which operates on irregularly nested sequences.

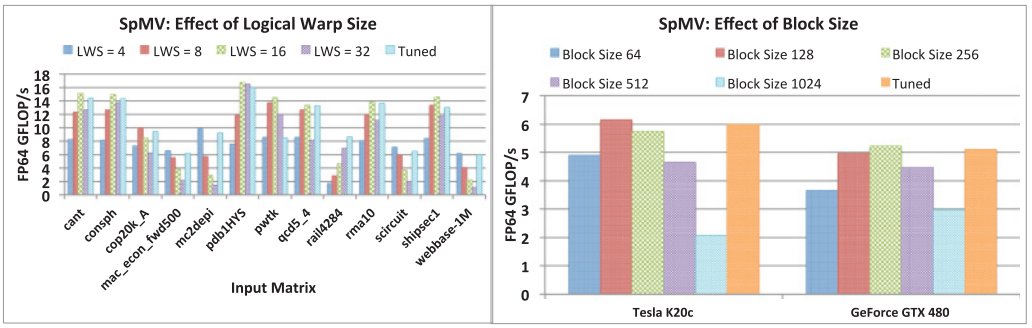


Fig. 6. CSR SpMV performance w.r.t. changing CUDA logical warp and block sizes.

However, as described by Ohshima et al. [2014], the OpenMP scheduling policy seems to affect SpMV performance only when the number of nonzeros is extremely high.

6.3. Performance Variation Among Code Variants

In the previous subsection, we demonstrated how Surge was capable of achieving performance that is on par with handwritten implementations. To better understand the role that autotuning plays in achieving this performance, we now analyze the relative performance of code variants with respect to different input datasets and target architectures. We focus on the SpMV benchmark on the GPU for this experiment and consider the `independent▷cooperative` schedule implemented using CUDA warps. We vary two parameters: logical warp size (LWS) and CUDA block size (`logical_warp_size` and `block_size_x` in Table IV); the remaining parameters (shown in the first row of Table V) are set to default values (`grain_size` and `block_size_y` to 1.)

Figure 6 (left) shows the effect of varying logical warp size with block size set to 128. Here, the x-axis represents matrices from the SpMV test set, and the y-axis shows performance in terms of throughput. Each bar represents a different LWS, and the bar labeled *Tuned* represents the autotuned version. As the graph shows, no single LWS is best across the test set, and the best LWS for a matrix depends on its average row length. Since the latter property is captured during autotuning (see Section 3.4), the tuned version chooses the correct LWS value in the majority of cases, with mispredictions typically resulting in the selection of the second-best LWS value. The only costly misprediction is for the `pwtk` matrix, which we believe is due to the model encountering a training matrix with similar nonzeros, but different average row length. Compared to warp size, the optimal block size primarily depends on the target architecture. Figure 6 (right) shows the effect of varying block size for the `webbase-1M` matrix on two distinct architectures: an NVIDIA Tesla K20c and an NVIDIA GeForce GTX 480. Here, the LWS is set to 4 (optimal value for this matrix) to ensure fair comparison. As the graph shows, on the K20c, a block size of 128 is optimal; this remains the case for the rest of the training set as well. However, on the GTX 480, the optimal block size turns out to be 256. In summary, the relative performance of code variants changes with different inputs and target architectures, and an autotuning system that can dynamically select the best variant based on input and architectural characteristics is essential for achieving optimal performance.

6.4. Productivity Gains

We provide a rough measure of productivity by counting the source lines of code required in Surge to express the *core computations* of our benchmarks (memory management and other bookkeeping code is not included) and comparing it with the number of lines required to express the same computation in the reference CUDA and OpenMP

Table VIII. Average Speedups over GPU and CPU Reference Implementations, and Source Lines of Code (SLOC) Required for Surge, and GPU and CPU Reference Implementations. SLOC for SpMV on the CPU Is Unknown as Intel MKL Is Closed-Source

Benchmark	Speedup		SLOC		
	GPU	CPU	Surge	GPU	CPU
`Reduction`	3.67	1.04	8	88	51
`Scan`	1.26	2.28	19	96	63
`SpMV`	1.17	0.93	9	55	unknown
`K-Means`	1.05	0.98	43	125	71
`CoMD`	1.01	0.95	71, 78	91	74

implementations. In the absence of a superior metric, we believe this captures the conciseness of Surge programs, while still maintaining readability. The last three columns of Table VIII show this comparison.

7. RELATED WORK

Nested Data-Parallelism. A majority of existing nested data-parallel programming models automatically employ the flattening transformation to convert nested data-parallel operations into flat data-parallel operations [Blelloch 1992; Chakravarty et al. 2007; Bergstrom et al. 2013], which may not be always optimal (see, for example, Keller et al. [2012]). This is especially true on architectures such as GPUs that expose a hierarchical parallelism structure. The notable exceptions are Copperhead [Catanzaro et al. 2011] and CuNesl [Zhang and Mueller 2012], which support compiling nested data-parallel operations to match the hierarchical parallelism available in GPUs. This mapping to hardware, however, is performed automatically by the CuNesl and Copperhead compilers and, unlike Surge (which exposes schedules and policies as part of the programming interface), no mechanism is exposed for experimentation with different mapping and implementation strategies. Lee et al. [2014] describe a framework for transforming nested data-parallel patterns encoded in a parametrized intermediate representation (IR) to efficient GPU code. A compiler analyzes the IR, generates a search space of possible implementations, and then prunes it before generating GPU-specific code. Similarly, Brown et al. [2016] describe an IR named DMLL that can be targeted from high-level parallel patterns (including nested data-parallel patterns). A compiler framework performs transformations on the IR to target distributed heterogeneous architectures. Since Surge exposes a relatively higher-level programming interface, such frameworks and IRs can potentially be targeted from Surge for more powerful analysis and code generation.

Decoupling Computation and Implementation. Outside the realm of nested data-parallel programming models, the concept of decoupling the specification of a computation from its implementation has been explored in the literature. Some flat data-parallel models such as Haskell Parseq [Jones and Singh 2009] and Thrust [Bell and Hoberock 2011] support the use of constructs such as `par` and `seq` to guide the evaluation of data-parallel operators. More recently, C++ extensions for parallelism [Hoberock 2016; Hoberock et al. 2016; Kretz 2016; Edwards et al. 2016] propose the use of *execution policies* and *executors* to control the execution of flat data-parallel operators. The Haskell REPA library [Keller et al. 2010] supports operators such as `map` and `foldAllP` and supports lazy evaluation on numeric arrays. The Galois system [Pingali et al. 2011] adopts a worklist-based approach to enable parallelization of irregular computations and supports the use of various decoupled runtime schedulers to process work items in parallel. Declarative task-based programming models such as Concurrent Collections (CnC) [Budimlić et al. 2010] decouple the high-level program task-graph from its hardware implementation. Computations at the task-level are then explicitly specified

using a number of different parallel programming models. Charm++ [Kale and Krishnan 1993] provides an asynchronous message-passing model to describe parallel programs. Halide [Ragan-Kelley et al. 2013] and Elixir [Prountzos et al. 2012] are domain-specific languages that enable users to decouple the specification of image processing pipelines and graph workloads, respectively, from their implementations using schedules. The design philosophy of systems such as Halide and Elixir is very similar to that of Surge; the actual definition of schedules, however, naturally differs. For instance, in Halide, the scheduling representation spans the search space of tradeoffs among locality, parallelism, and redundant recomputation in stencil pipelines. The Delite domain-specific language compiler framework [Brown et al. 2011] uses Lightweight Modular Staging [Rompf and Odersky 2010] to build an intermediate representation that can represent both parallel patterns and domain-specific constructs. The Delite compiler then compiles parts of the IR to Scala, C++, or CUDA. Similarly, the Lime [Auerbach et al. 2010] compiler generates Java code for the entire program, plus OpenCL for GPUs and Verilog for FPGAs; the Liquid Metal Runtime [Auerbach et al. 2012] then selects which compiled code to use. Systems such as OpenMP [Dagum and Menon 1998] and OpenACC [NVIDIA et al. 2015] and loop transformation frameworks provide directive-based approaches to parallelize sequential code. Our work, in contrast, is specifically focused on nested data parallelism. Existing systems such as the ones described above, however, need not be mutually exclusive with Surge. For example, languages such as CnC define an entirely separate coordination language within which the programmer describes the data-parallel computation. A combined system can make use of Surge to provide the finer-grained data parallelism.

Programming Models Supporting Autotuning. Recent work has explored the integration of autotuning into parallel programming models. In Tangram [Chang et al. 2016], expert programmers specify a spectrum of *codelets*, and the Tangram compiler composes them to generate new ones; the best codelet is then chosen through autotuning. While both Surge and Tangram target performance portability, Surge exposes a functional interface consisting of high-level operators and collections for expressing computations that does not require expert knowledge to generate high-performance code. Steuwer et al. [2015] describe a system that transforms high-level functional expressions into OpenCL code using a set of rewrite rules. By exploring the space of rewrite rules, multiple implementations are generated and autotuned. While the data-parallel operators and the idea of using autotuning to find a suitable implementation are similar to that of Surge, there are important differences between the two systems: First, the rewrite rules in the Steuwer framework target only OpenCL, with the assumption that OpenCL can be used to provide performance portability across current and future architectures; Surge, in comparison, is not restricted to any single platform. Also, a new set of rewrite rules must be written by an expert programmer to target every new platform in the Steuwer framework; Surge, however, greatly simplifies this process by abstracting the dependence structure of operators in a platform-agnostic way, as described in Section 4.

8. CONCLUSIONS

This article has presented Surge, a new nested data-parallel programming system. The Surge programming interface, implemented purely as a C++11 library, decouples high-level specification of computations from low-level implementation details. The code generation and autotuning subsystems use this two-level mechanism to systematically generate code for multiple platforms and tune it with respect to both the architecture and input dataset. For five benchmarks expressed in Surge, we describe how high-performance implementations for GPUs and CPUs are automatically generated and demonstrate performance that is on par with or better than handcrafted reference

implementations. With the initial framework in place, we plan to add support for more platforms and also explore compiler-based techniques for operator transformations.

APPENDIX

Listing 8 shows the full source code for the SpMV benchmark, including all headers and declarations.

```
#include <iostream>
#include <ctime>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <surge/sequences/sequence.h>
#include <surge/operators/map.h>
#include <surge/operators/gather.h>
#include <surge/operators/reduce.h>
#include <surge/operators/nest.h>
// CUSP matrix I/O routines
#include "matrix_io.h"
// The code generator and autotuner communicate
// with the main application via the following
// auto-generated file (surge_config.h).
#include "surge_config.h"
// The behavior of the execute() function
// changes based on what is defined in
// surge_config.h. So include it later.
#include <surge/operators/execute.h>

using value_t = double;
// Use Thrust's device_vector for CUDA. On the CPU, change to thrust::host_vector.
template<typename T>
using vector_t = thrust::device_vector<T>;

int main(int argc, char *argv[])
{
  using namespace surge;

  if(argc != 2) {
    std::cerr << "Usage: spmv matrix_market_file\n";
    exit(-1);
  }
  // Use CUSP I/O routines to read matrix market file
  using csr_matrix_t = cusp::csr_matrix<int, value_t, cusp::host_memory>;
  csr_matrix_t matrix;
  read_mtx_as_csr(argv[1], matrix);

  using index_t = typename csr_matrix_t::index_type;
  using value_t = typename csr_matrix_t::value_type;
  const int n_cols = matrix.num_cols;
  const int n_rows = matrix.num_rows;

  // Create vector on host and populate with random values.
  thrust::host_vector<value_t> h_vector(n_cols, value_t(0));
  srand(time(NULL));
  thrust::generate(h_vector.begin(), h_vector.end(),
  []() { return value_t(std::rand()) / value_t(RAND_MAX); });

  // Allocate memory on and copy data to target device.
  vector_t<index_t> rows(matrix.row_offsets.begin(), matrix.row_offsets.end());
  vector_t<index_t> cols(matrix.column_indices.begin(), matrix.column_indices.end());
  vector_t<value_t> vector(h_vector);
  vector_t<value_t> nzs(matrix.values.begin(), matrix.values.end());
  vector_t<value_t> result(n_rows, value_t(0));

  // Create views (sequences) for Surge operators.
  auto s_vector = make_sequence(vector.begin(), vector.end());
  auto s_row_offsets = make_sequence(rows.begin(), rows.end());
```

```
60    auto s_nonzeros = make_sequence(nzs.begin(), nzs.end());
61    auto s_column_indices = make_sequence(cols.begin(), cols.end());
62    auto s_result = make_sequence(result.begin(), result.end());
63
64    // Create nested sequences for matrix and column indices.
65    auto s_matrix = nest(s_nonzeros, s_row_offsets);
66    auto s_indices = nest(s_column_indices, s_row_offsets);
67
68    using row_t = decltype(s_matrix[0]);
69    using index_row_t = decltype(s_indices[0]);
70
71    auto spmv =
72      // Apply dot product across all rows of matrix.
73      // SURGE_LAMBDA_PREFIX expands to __host__ __device__ when compiling with NVCC.
74      map([=] SURGE_LAMBDA_PREFIX (row_t row, index_row_t indices) {
75            auto mul = [](value_t x, value_t y) { return x*y; };
76            auto plus = [](value_t x, value_t y) { return x+y; };
77            // Gather elements from vector s_vector
78            auto z = gather(s_vector, indices);
79            // Element-wise multiplication vector with row
80            auto vector_mul = map(mul, row, z);
81            // Sum up elements to obtain dot product
82            return reduce(plus, vector_mul, value_t(0));
83         },
84         s_matrix, s_indices);
85
86    // Realize SpMV computation
87    execute(spmv, s_result,
88            // Target CUDA. Change to backend::omp for OpenMP.
89            backend::cuda{}
90           );
91    // Use results.
92  }
```

Listing 8. Full Surge code for SpMV. The platform has been set to CUDA in this example.

ACKNOWLEDGMENTS

We thank NVIDIA Corporation for generous equipment donations and members of the NVIDIA research group for valuable discussions. We also thank Bryan Catanzaro for contributing to the initial design of Surge.

REFERENCES

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. ACM, 303–316.

Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. 2012. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*. ACM, 271–276.

Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*. ACM, 89–108.

Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. ACM, Article 18.

Nathan Bell and Jared Hoberock. 2011. Thrust: A productivity-oriented library for CUDA. *GPU Comput. Gems Jade Ed.* 2 (2011), 359–371.

Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. 2013. Data-only flattening for nested data parallelism. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, 81–92.

Guy E. Blelloch. 1992. *NESL: A Nested Data-Parallel Language*. Technical Report CMU-CS-95-170. Carnegie Mellon University, Pittsburgh, PA.

Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. 2016. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO 2016)*. ACM, 194–205.

Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 20th Parallel Architectures and Compilation Techniques Conference (PACT'11)*. ACM, 89–100.

Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşirlar. 2010. Concurrent collections. *Sci. Program.* 18, 3–4 (Aug. 2010), 203–217.

Bryan Catanzaro. 2014. GPU K-Means Clustering. Retrieved October 28, 2016 from https://github.com/bryancatanzaro/kmeans.

Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2011. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, 47–56.

Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data parallel haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP'07)*. ACM, 10–18.

Li-Wen Chang, Izzat El Hajj, Christopher Rodrigues, Juan Gómez-Luna, and Wen-mei Hwu. 2016. Efficient kernel synthesis for performance portable programming. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*.

Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry standard API for shared-memory programming. *IEEE Comput. Sci. Eng.* 5, 1 (1998), 46–55.

Steven Dalton, Nathan Bell, and Michael Garland. 2010. CUSP Library. Retrieved October 28, 2016 from http://cusplibrary.github.io/.

Tim Davis. 2011. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.* 38 (2011), 1:1–1:25. Issue 1.

Denis Demidov, Karsten Ahnert, Karl Rupp, and Peter Gottschling. 2016. VexCL Symbolic Type. Retrieved October 28, 2016 from http://vexcl.readthedocs.io/en/latest/symbolic.html.

H. Carter Edwards, Christian Trott, Juan Alday, Jesse Perla, Mauro Bianco, Robin Maffeo, Ben Sander, and Bryce Lelbach. 2016. Polymorphic multidimensional array reference. *ISO/IEC C++ Standards Committee Paper P0009R2*. Retrieved October 28, 2016 from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0009r2.html.

ExMatEx. 2015. DoE Exascale Co-Design Center for Materials in Extreme Environments. Retrieved October 28, 2016 from http://www.exmatex.org.

Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. 2009. Annotation-based empirical performance tuning using orio. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'09)*. IEEE Computer Society, 1–11.

Jared Hoberock. 2016. Working draft, technical specification for C++ extensions for parallelism version 2. *ISO/IEC C++ Standards Committee Paper N4578* (2016). Retrieved October 28, 2016 from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4578.html.

Jared Hoberock, Michael Garland, and Olivier Giroux. 2016. An interface for abstracting execution. *ISO/IEC C++ Standards Committee Paper P0058R1* (2016). Retrieved October 28, 2016 from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0058r1.pdf.

Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Comput. Surv.* 28, 4es, Article 196 (Dec. 1996).

Intel. 2016. Math Kernel Library. Retrieved October 28, 2016 from https://software.intel.com/en-us/intel-mkl.

Simon Peyton Jones and Satnam Singh. 2009. A tutorial on parallel and concurrent programming in haskell. In *Proceedings of the 6th International Conference on Advanced Functional Programming (AFP'08)*. Springer-Verlag, 267–305.

Laxmikant V. Kale and Sanjeev Krishnan. 1993. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the 8th Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'93)*. ACM, 91–108.

Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. 2012. Vectorisation avoidance. In *Proceedings of the 2012 Haskell Symposium (Haskell'12)*. ACM, 37–48.

Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. 2010. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*. ACM, 261–272.

Matthias Kretz. 2016. Data-parallel vector types and operations. *ISO/IEC C++ Standards Committee Paper P0214R1* (2016). Retrieved October 28, 2016 from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0214r1.pdf.

HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Tiark Rompf, and Kunle Olukotun. 2014. Locality-aware mapping of nested parallel patterns on GPUs. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, 63–74.

Stuart P. Lloyd. 1982. Least squares quantization in PCM. *IEEE Trans. Inform. Theor.* 28, 2 (Mar. 1982), 129–137.

Duane G. Merrill, III. 2011. *Allocation-oriented Algorithm Design with Application to GPU Computing*. Ph.D. Dissertation. University of Virginia, Charlottesville, VA. UMI Order Number: AAI 3501820.

Saurav Muralidharan, Manu Shantharam, Mary Hall, Michael Garland, and Bryan Catanzaro. 2014. Nitro: A framework for adaptive code variant tuning. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'14)*. IEEE Computer Society, 501–512.

NVIDIA, Cray, CAPS, and PGI. 2015. The OpenACC Specification version 2.0a. Retrieved October 28, 2016 from http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.

Shigetoshi Ohshima, Takahiro Katagiri, and Morio Matsumoto. 2014. Performance optimization of SpMV Using CRS format by considering OpenMP scheduling on CPUs and MIC. In *Proceedings of the 8th IEEE International Symposium on Embedded Multicore/Manycore SoCs (MCSoc)*. 253–260.

Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, 12–25.

Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. 2012. Elixir: A system for synthesizing concurrent graph programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. ACM, 375–394.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, 519–530.

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'10)*. ACM, 127–136.

Nikolay Sakharnykh. 2013. CoMD-CUDA. Retrieved October 28, 2016 from https://github.com/NVIDIA/CoMD-CUDA.

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP'15)*. ACM, New York, NY, 205–217.

Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. 2009. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*. 1–12.

Vladimir N. Vapnik. 1998. *Statistical Learning Theory*.

Todd Veldhuizen. 1995. Expression templates. *C++ Report* 7, 5 (1995), 26–31.

Yongpeng Zhang and F. Mueller. 2012. CuNesl: Compiling nested data-parallel languages for SIMT architectures. In *Proceedings of the 41st International Conference on Parallel Processing (ICPP)*. 340–349.

Received June 2016; revised September 2016; accepted October 2016