Merge: A Programming Model for Heterogeneous Multi-core Systems

Michael D. Linderman[1] * Jamison D. Collins[2] Hong Wang[2] Teresa H. Meng[1]

[1]Dept. of Electrical Engineering
Stanford University
Stanford, CA, USA

[2]Microarchitecture Research Lab
Intel Corporation
Santa Clara, CA, USA

{mlinderm,thm}@stanford.edu {hong.wang,jamison.d.collins}@intel.com

Abstract

In this paper we propose the Merge framework, a general purpose programming model for heterogeneous multi-core systems. The Merge framework replaces current ad hoc approaches to parallel programming on heterogeneous platforms with a rigorous, library-based methodology that can automatically distribute computation across heterogeneous cores to achieve increased energy and performance efficiency. The Merge framework provides (1) a predicate dispatch-based library system for managing and invoking function variants for multiple architectures; (2) a high-level, library-oriented parallel language based on map-reduce; and (3) a compiler and runtime which implement the map-reduce language pattern by dynamically selecting the best available function implementations for a given input and machine configuration. Using a generic sequencer architecture interface for heterogeneous accelerators, the Merge framework can integrate function variants for specialized accelerators, offering the potential for to-the-metal performance for a wide range of heterogeneous architectures, all transparent to the user. The Merge framework has been prototyped on a heterogeneous platform consisting of an Intel Core 2 Duo CPU and an 8-core 32-thread Intel Graphics and Media Accelerator X3000, and a homogeneous 32-way Unisys SMP system with Intel Xeon processors. We implemented a set of benchmarks using the Merge framework and enhanced the library with X3000 specific implementations, achieving speedups of 3.6x – 8.5x using the X3000 and 5.2x – 22x using the 32-way system relative to the straight C reference implementation on a single IA32 core.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.4 [*Programming Langauges*]: Processors—Compilers

General Terms Performance, Design, Languages

Keywords heterogeneous multi-core, GPGPU, predicate dispatch

1. Introduction

Mainstream processor designs will continue to integrate more cores on a single die. For such architectures to scale low energy-per-instruction (EPI) cores are essential (10; 2). One approach to improving EPI is heterogeneous multi-core design in which cores vary in functionality, performance, and energy efficiency. These systems often feature instruction sets and functionality that significantly differ from general purpose processor cores like IA32, and for which there is often limited or no sophisticated compiler support. Consequently developing applications for these systems is challenging and developer productivity is low. At the same time, the exponential growth in digital information is creating many workloads, termed *informatics applications*, which require computational performance that cannot be supplied by conventional architectures (18; 7). There is an urgent need to assist the creators of these informatics workloads in taking advantage of state-of-the-art heterogeneous multi-core systems.

Several programming frameworks, termed *domain specific virtual machines* (DSVM) (*e.g.,* RapidMind (21), PeakStream (25), Accelerator (29)), have been developed to compile high level data parallel languages or libraries to specialized accelerators (*e.g.* SSE extensions, GPUs, and the CELL processor SPEs (26)). These approaches provide a data parallel API that can be efficiently mapped to a set of vector and array arithmetic primitives that abstract away the differences in the ISAs or APIs of the target platforms. While the vector and array primitives are appropriate for the SIMD and SPMD functionality of the above architectures, they will be inadequate for representing more complex parallel functionality, *e.g.* the MIMD operations supported by various heterogeneous systems.

EXOCHI (31) enables the creation of an API for heterogeneous architectures, even those without significant compiler support, by inlining accelerator-specific assembly or domain specific language into traditional C/C++ functions. The resulting C/C++ code is largely indistinguishable from existing ISA intrinsics (like SSE), enabling the programmer to create new intrinsics, termed *function-intrinsics*, of any complexity for any architecture that supports the lightweight EXO interface. The function-intrinsics form a set of primitives, similar to those in the DSVMs. But, unlike the DSVMs, the primitives are not restricted to vector or array operations, and can abstract any ISA or API. EXOCHI is designed to enable the creation of accelerator-specific function-intrinsics; it does not attempt to determine if or when a particular intrinsic should be invoked.

Accelerators are typically optimized for specific data patterns, and will either be unusable or perform poorly when computations that fall outside the optimized regime. GPUs, for example, are particularly sensitive to the number of threads that are created, and to the ability to vectorize those threads. One too many threads (*e.g.*, 33 threads for an SMT architecture that supports 32 thread contexts) or a change in vector length (*e.g.*, 8 to 12) will result in an increase in execution time that is disproportionate to the increase in workload

* Work performed while at Intel

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA
Copyright © 2008 ACM 978-1-59593-958-6/08/0003. . . $5.00

size. Generalizing an algorithm can be difficult, with different data patterns often requiring different implementations, or even favoring different accelerators. Since the data pattern often cannot be determined statically, the problem of selecting and scheduling the right implementation is inherently dynamic. As a result, the DSVMs utilize runtime compilation to produce code for a specific data pattern for a statically determined accelerator. However, achieving the best performance or EPI for a heterogeneous system requires more than selecting or generating the code for a given accelerator; equally as important is judiciously distributing work among the available heterogeneous cores.

In this paper we present the **Merge** framework, a high-level parallel programming model, compiler and runtime for heterogeneous multi-core platforms. The Merge framework maps an application to a user-extensible set of primitives in the form of function-intrinsics (created with EXOCHI and other tools). Our focus is not to just support general purpose computation on specialized hardware, but to truly exploit *heterogeneous systems*, achieving increased performance by dynamically choosing function-intrinsics to utilize all available processors. This paper makes the following contributions:

- We describe a dynamic framework that automatically distributes computations among different cores in a heterogeneous multi-core system, in contrast to static approaches, or dynamic approaches that only target the accelerator.
- We describe an extensible framework that enables new architectures to be readily integrated into and exploited by existing programs.
- We present an implementation of the Merge framework compiler and runtime, and report significant performance gains for heterogeneous (Intel Core 2 Duo processor and an 8-core 32-thread Intel Graphics Media Accelerator X3000) and homogeneous (Unisys 32-way SMP with Intel Xeon processors) platforms.

The rest of the paper is organized as follows; Sections 2 and 3 present the Merge framework. Section 4 details a performance evaluation. Section 5 reviews related work and Section 6 concludes.

2. Library-Based Programming Model

The Merge framework uses EXOCHI (31) to create and implement APIs across a wide range of heterogeneous architectures. A sketch of the operation of EXOCHI is shown in Fig. 1. EXO supports a shared virtual memory multi-threaded programming model for heterogeneous systems. The heterogeneous cores are exposed to the programmer as sequencers (12), on which user-level threads, encoded in the accelerator-specific ISA, execute. Using *C for Heterogeneous Integration* (CHI), the programmer implements function-intrinsics for these sequencers by inlining accelerator-specific assembly and domain-specific languages into traditional C/C++ code. The Merge framework uses the C/C++ function intrinsics, executing on the abstract sequencers, as its primitives. The C/C++ intermediate language and uniform shared virtual memory abstraction enabled by EXOCHI provide a powerful foundation on which to build a expressive and extensible programming model that facilitates cooperative execution among multiple, different processors.

Merge applications are expressed as a hierarchical set of functions that break the computation into successively smaller operations to expose parallelism and describe data decomposition. An example is shown in Fig. 2 for a part of the k-means clustering algorithm. K-means iteratively determines the set of k clusters that minimize the distance between each observation and its assigned cluster. In this portion of k-means, the clusters for the next iteration are computed by averaging all of the observations assigned to a given cluster. Numerous decompositions are possible, shown as *granularity variants* in the figure, the choice of which affects the amount and granularity of the parallelism. At any granularity

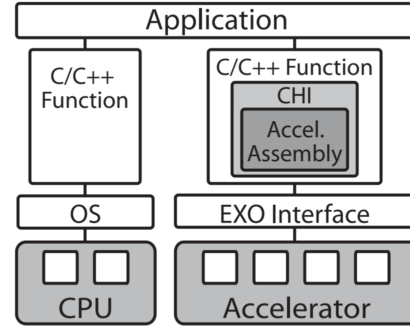

Figure 1. Sketch of EXOCHI framework.

and at any step in the hierarchy, the application can target existing library APIs, which might already have multiple implementations for heterogeneous accelerators, shown as *target variants* in the figure, or define new APIs that later might be extended with accelerator-specific function-intrinsics. Each function call is thus potentially a choice between continuing the decomposition, or invoking a function-intrinsic that maps the parallelism in the algorithm to a specific accelerator architecture.

The combination of different decompositions, and multiple target variants provides a rich set of potential implementations from which the Merge compiler and runtime can choose to optimize application performance and efficiency. Choosing the best function variant has three components: narrowing variants to those for the cores that are available in a given platform; choosing a variant so that the work is distributed among the available processors; and choosing the best parameterization for a given function and architecture.

In existing library-based approaches the first choice is primarily static, and is typically tackled via selective compilation (*i.e.,* `#ifdef`). The second choice is primarily dynamic, and is typically tackled with an ad hoc work queue, specific to the given processors and functions. The third choice is a combination of static and dynamic (depending on whether data patterns are known *a priori*) and is typically tackled through a combination of selective compilation and manual dispatch using `if-else` statements. When there were only a few target architectures available, the above approaches were less problematic, but as the diversity of heterogeneous systems grows, and systems with multiple accelerators become commonplace, the combinatorics will make the above techniques for selecting the best function variant untenable. Even with current library management tools (reviewed in Section 5), users often need to keep track of all of the function variants available, create manual dispatch code, and maintain multiple platform-specific versions of each program, all of which make applications non-portable and difficult to maintain.

The Merge framework replaces the current ad hoc mix of static and dynamic variant selection with a unified dynamic approach using predicate dispatch (8) and an architecture-agnostic work queue. Programmers annotate function variants with predicates specifying the target architecture, and any invariants required for correct execution. The predicates are automatically translated into dispatch conditionals that eliminate the need for the selective compilation or manual dispatch code described above, which facilitates reuse and makes applications easier to maintain. At runtime, the dispatch conditionals can be used in conjunction with the task queue to dispatch work to specific processors as they become idle in order to distribute computation across heterogeneous cores. New target variants can be added simply by creating new implementations that share the common interface.

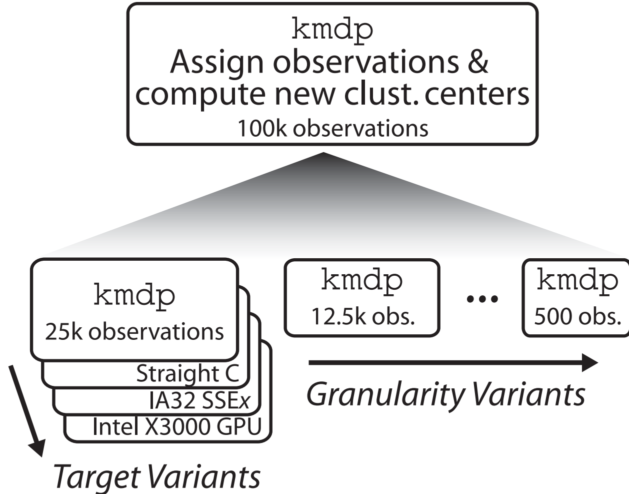

Figure 2. Hierarchical decomposition of k-means algorithm using functions with different granularities and target architectures.

The familiar shared memory abstraction provided by EXOCHI facilitates the adaption of existing scheduling techniques and implementations for use with heterogeneous systems. Since all of the processors, general-purpose CPUs and specialized accelerators, execute in the same virtual address space, each can potentially play a role in the computation. The uniform abstraction minimizes the amount the architecture-specific infrastructure required within the Merge compiler/runtime to enable that cooperation. As a result the Merge framework can exploit many different heterogeneous systems using a single, architecture-agnostic scheduler. The architecture-agnosticism simplifies the integration of new processor types into Merge-targeted systems. All that is required is a lightweight EXOCHI interface for the accelerator, an architectural identifier for use with the predicate dispatch, and a single function variant for that accelerator. The new variant will automatically be incorporated into the dispatch system (as described above) and invoked when appropriate by Merge applications.

3. Merge Framework Design

The Merge framework consists of three components: (1) a high-level parallel programming language based on the map-reduce pattern; (2) a predicate-based library system for managing and invoking function variants for multiple architectures; and (3) a compiler and runtime which implement the map-reduce pattern by dynamically selecting the best available function variant. Fig. 3 shows a sketch of the operation of the Merge framework.

Although the Merge framework predicate dispatch system and runtime can stand alone, we believe they are more effective when used with a high-productivity, high-level parallel programming language similar to those provided by the DSVMs. The Merge high-level language uses the map-reduce pattern, which underlies Google's MapReduce (6) and is present in LISP and many other functional languages. The Merge compiler directly translates the explicit parallelism of the map-reduce pattern to independent work units, represented by the small boxes in the figure, that can be mapped to available function variants and dispatched to the processors cores. Similar to the Sequoia programming language (9), each work unit, the principal parallel construct in the Merge framework, is a *task*, a side-effect free function. The task construct specifies the invariants that must be maintained by a function variant in order to support a correct and efficient mapping between the front-end language and the function library.

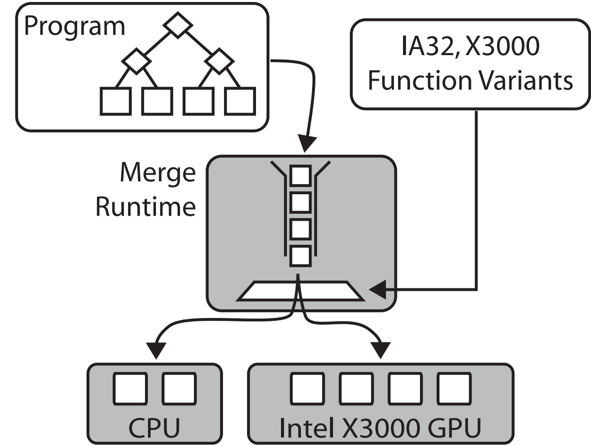

Figure 3. Sketch of Merge framework. The program, written using the map-reduce pattern, is transformed into work units which are pushed onto the task queue. Work units are dynamically dispatched from the queue, distributing the work among the processors that have applicable function variants.

All function variants in the library are considered tasks, and are restricted to the environment defined by their arguments (*i.e.*, do not interact with any global variable). Tasks can directly communicate with other tasks only through creating subtasks or returning to the parent task. Library functions are required to satisfy the task semantics to ensure that each invocation of given variant (with disjoint arguments), such as might occur in a map operation, is independent and can be executed concurrently. Unlike Sequoia, however, sibling tasks may communicate indirectly though their arguments and a task can have dependent tasks other than its direct parent. In this way dataflow dependencies between tasks can be enforced without spawning.

3.1 Map/Reduce Parallel Pattern

Informatics application, which often feature the integration or reduction of large datasets, are well described by the map-reduce pattern. In k-means for example, the algorithm reduces the large number of data observations belonging to one or more discrete classes to a small set of clusters that summarize those classes. All computations are decomposed into a set of map operations and a reduce operation, with all map operations independent and potentially concurrent. The decomposition can be applied hierarchically across granularities, from single operations such as the multiplies in an inner product, to complex algorithms. An example implementation for a portion of k-means using the map-reduce pattern in is shown in lines 1-11 of Fig. 4.

Each `mapreduce` call has three parts: (1) the generator(s) to drive the map operation by creating a task tuple space, (2) the map function and (3) the optional reduce function(s). The map-reduce primitives are shown in Table 1. The generators work in concert with collections which provide data decomposition and task mapping. The collection provides the mapping between a task's tuple space and its data. A collection type does not actually contain any data, just the necessary information to provide the tuple mapping. In lines 9-11 of Fig. 4 the function `dp` is mapped over the rows of `dp` and elements of `assgn` (indicated by the array subscript), reducing the arrays `ccnew` and `hist` produced by each map invocation. The tuple space is the row indices of `dp` and is created by the generator in line 9. `Array2D`, one of the basic collections supplied in the Merge framework, wraps a 2D matrix and is iterable by rows (similarly, `Array1D` wraps a vector and is iterable by elements).

```
1  bundle km0 {
2   void kmdp(Array2D<float> dp,
3             Array2D<float> cc,
4             Array1D<int> assgn,
5             Array2D<float> ccnew,
6             Array1D<int> hist) {
7     // Assign observations and compute new cluster
8     // centers for next iteration in parallel
9      mapreduce(int i=0; i < dp.rows; i++)
10       kmdp(dp[i],cc,assgn[i],
11            red<sum>(ccnew),red<sum>(hist));
12 }};
13
14 bundle km1 : public Seq {
15  void kmdp(Array2D<float> dp,
16            Array2D<float> cc,
17            Array1D<int> assgn,
18            Array2D<float> ccnew,
19            Array1D<int> hist) {
20    // Dispatch control predicate
21    predicate(dp.rows < 5000);
22
23    // ... Sequential implementation ...
24    for (int i=0; i < dp.rows; i++) {
25     float dist, min = FLT_MAX; int idx;
26     for (int k=0; k < cc.rows; k++) {
27      dist = euclidean_distance::call(dp[i],cc[k]);
28      if (dist < min) { min = dist; idx = k; }
29     }
30     sum::call(ccnew[k],dp[i]); sum::call(hist[k],1);
31 }}};
32
33 bundle km2 : public X3000 {
34  void kmdp(Array2D<float> dp,
35            Array2D<float> cc,
36            Array1D<int> assgn,
37            Array2D<float> ccnew,
38            Array1D<int> hist) {
39      // Dispatch control predicate
40      predicate(dp.rows < 5000);
41      predicate(dp.cols == 8 && cc.rows == 4);
42      predicate(arch == X3000_arch);
43
44      // ... X3000 Implementation using EXOCHI ...
45 }};
```

Figure 4. Example implementation for portion of k-means algorithm; `km0::kmdp` assigns observations to a cluster and computes new cluster centers for an array of observations in parallel. `km1::kmdp` and `km2::kmdp` are variants of `kmdp` targeting traditional CPUs and the X3000 GPU.

3.2 Function Library Manager

The function library in a Merge framework program is treated as a pool of target variants that can be bundled into *generic functions* – the collection of functions of the same name, number of arguments and result types (22). The generic functions are used in place of a particular variant to provide a choice of implementations to the runtime. Annotations are supplied with each function, and automatically processed at compile time by the library manager, called the *bundler*, to produce a set of meta-wrappers. The syntax for a function variant and the predicate annotations is shown in Table 1. The meta-wrappers implement the generic function and invoke the *most-specific applicable function* (defined in more detail below) belonging to that generic function. Automatically generating the meta-wrappers enables new implementations to be integrated into a given generic function without manually modifying pre-existing wrappers, or creating new ones (similar to multiple dispatch in object-oriented languages).

Table 1. Merge syntax for function bundling, dispatch control predicates and map-reduce. The literal non-terminals in the predicate syntax are constants of the specified type, while *Identifier* is any variable or field in scope (syntax adapted from (22)).

Bundling

```
bundle dp0 : public sse {
  predicate(...); void dp(..); };
```

Define a new variant for `dp` with fully qualified name `dp0::dp` belonging to the group `sse` with a dispatch control predicate.

Predicates

pred	::=	`arch` | *lit* | *tgt* | *uop pred* | *pred bop pred*
lit	::=	*integerLiteral* | *boolLiteral* | *floatingpointLiteral*
tgt	::=	`this` | *Identifier* | *tgt.Identifier*
uop	::=	∼ | −
bop	::=	`&&` | `||` | `==` | `!=` | `<` | `>` | `>=` | `<=` | − | + | ⋆

Map/Reduce

`seqreduce(int i=0; i < n; i++) { ... }`

Execute a function over elements of a collection sequentially, potentially performing a reduction one or more arguments. Reduction arguments are described with `reduce<reducefn>(argument)`, indicating the function to be used in the combining. Multiple `seqreduce` statements can be nested, with the inner statement body only containing a function call.

`mapreduce(int i=0; i < n; i++) { ... }`

Execute a function over elements of a collection in parallel, potentially performing a reduction one or more arguments. Reduction can implemented with a parallel tree.

The annotations, which are inspired by Roles (17), the annotation language described in (11) and Fortress (1), are of two types, predicates and groups. Predicates take the form of logical axioms, and typically include constraints on the structure or size of inputs. For example, additional variants for `dp` are shown Fig. 4 lines 14-45, including an X3000-specific GPU implementation. The X3000 supports 8-wide SIMD execution, so this particular variant is limited to observation vectors of length 8, and as a result of loop unrolling, four cluster centers (predicates on line 41). Groups enable hierarchical collections of variants, similar to classes with single inheritance, and are typically used to express the accelerator hierarchy. For example a system may have groups for generic, sequential, SSE, and X3000 (shown on lines 14 and 33).

Similar to the implementation in (22), predicates in the Merge framework must resolve to a Boolean and may include literals, enumerations, access to arguments and fields in scope, and a set of basic arithmetic and relational operators on integers and floating point values (all arithmetic expressions must be linear). A function variant is applicable if its predicates evaluate to `true` for the actual arguments at runtime. All the variants belonging to a given generic function must be exhaustive, in that there is an applicable variant for the entire parameter space (evaluated at compile time). Exhaustiveness checking is performed in the context of groups. All of the variants in a group, as well as all variants in its parent, grandparent, *etc.*, are considered when checking for exhaustiveness.

A variant is the most specific if overrides all other applicable variants. Variant m_1 overrides m_2 if m_1 is in a subgroup of m_2 and the predicates of m_1 logically imply the predicates of m_2. Unlike more object oriented predicated dispatch systems, variants do not need to be unambiguous. As described earlier, it is assumed that many variants will be applicable, although not uniquely more specific, over a subset of the parameter space, and a secondary ordering mechanism, such as compile time order or profiling will be used to determine dispatch order. Similar to exhaustiveness checking, ordering analysis is performed in the context of groups. Variants are ordered within their group, such that any applicable variant in a group would be invoked before any variants in its parent

group. In this way, high performance GPU variants on the X3000 would be invoked before defaulting to a more generic CPU-based implementation.

The product of the bundler is a set of three meta-wrappers (call, lookup, and speculative lookup) for each function. The call wrapper simply calls the selected variant directly, while the lookup wrapper returns a function pointer to the variant. Both the call and lookup wrappers require that suitable variants exist and are exhaustive; speculative lookup does not have such a requirement and will support the lookup of variants that do not exist, simply returning null if no suitable variant is available. The speculative functionality enables the runtime to lookup potentially higher performing variants which may not exist, before defaulting to more general implementations.

3.3 Compiler and Runtime

The Merge framework compiler converts the map-reduce statements to standard C++ code which interfaces with the runtime system. Conversion consists of three steps: (1) translating the possibly nested generator statements into a multi-dimensional blocked range which will then drive execution; (2) inserting speculative function lookups to retrieve the variants for the map and reduce operations; and (3) inserting runtime calls to invoke the retrieved variants as appropriate.

The blocked range concept is drawn from the Intel Threading Building Blocks (14) and describes a one or more dimensional iteration space that can be recursively split into progressively smaller regions. The depth of the recursion determines the extent of the parallelism, with each split creating two potentially concurrent computational tasks. The generator statements are directly mapped to a blocked range (as a result generator statements are limited to those which can be directly mapped to a contiguous range at compile time), with the blocked range serving as the index for task identification, and iteration through input and output collections.

The dataflow of a map-reduce pair forms an implicit tree with the map operations at the leaves and the reduce operation at the joins. Each leaf represents a potentially independent task that can be mapped to any of the applicable function variants. A *unit-node* function, comprising a single leaf or join, can represent too fine a granularity, however, so for multi-core architectures there is particular interest in *multi-node* functions, which encompass multiple leaves or joins (alternately described as encompassing a non-unit size blocked range). Invoking a multi-node variant effectively short-circuits the map-reduce structure, deferring instead to the function. Thus with a multi-node function the author can explicitly transform the generic map-reduce parallelism, which might otherwise be implemented as a set of unit-node tasks, to custom parallelism for a specific architecture.

Multi-node functions are preferentially selected over unit-node functions, as they potentially offer a custom parallel implementation for a specific architecture. Most of the X3000 function variants used in this paper are implemented as multi-node variants. The computation in a unit-node is often insufficient to create the necessary threads for efficient execution, while the complete function (because of reductions or other constructs) is not easily implemented completely on the GPU (*i.e.,* as a unit-node variant for the parent). In these cases the variant is cast as a multi-node function (with the unit-nodes as threads), that will be invoked for a portion of the map-reduce nodes.

The ordering of function lookups and resulting actions at runtime is shown in the flowchart in Fig. 5. Based on the results of the speculative lookup for multi-node variants spanning all the map and reduce operations one of three execution scenarios is possible: all functions have multi-node variants; at least one, but not all, functions have multi-node variants; or no functions have multi-

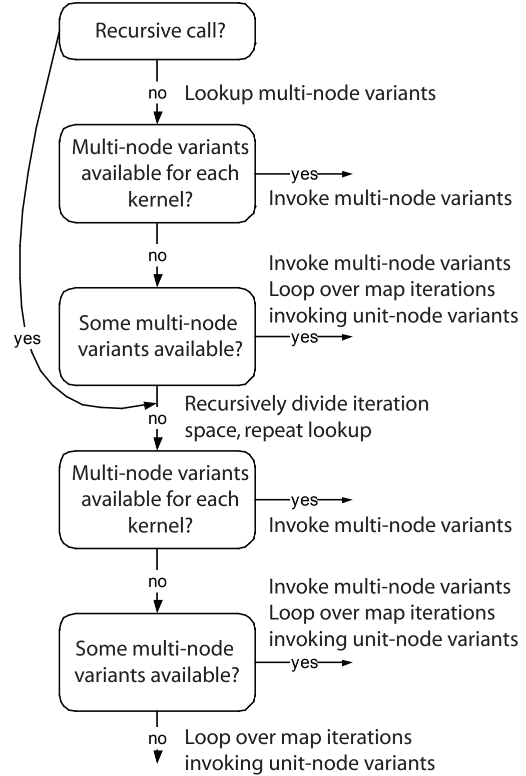

Figure 5. Runtime variant selection and invocation flowchart

node variants. In the first two cases the multi-node intermediate collection is allocated, providing space for the results of the the map operations (or alternatively the input to the reduction operation). In the third only the unit-node intermediate collection is required. When available the multi-node variants are invoked directly. Otherwise the blocked range is recursively split to drive parallel execution. After recursive splitting, the tuple space is divided into a set of non-overlapping blocks, each representing a subset of the problem. These blocks form the work units that are pushed onto the task queue for distribution to the different cores.

The lookup procedures are repeated when a task is issued at runtime from the queue, as shown in the flowchart. The per-task lookup enables dynamic workload balancing; as each processor becomes available, tasks, executing variants specific to that architecture, can be dispatched. The additional multi-node lookup (before defaulting to the unit-node variant) allows multi-node variants that encompass the entire block of nodes assigned to that task to be selected. Returning to `kmdp` in lines 1-12 of Fig. 4 to summarize the lookup procedures, the map statement indicates that this function is implemented by invoking a variant of `kmdp`, which takes a observation vector, over all observations in parallel. The Merge runtime will consider three different variants implementing `kmdp` from that representation: a multi-node variant spanning all of the observations, a multi-node variant spanning a subset of the observations, and unit-node variant spanning one observation.

3.4 Compiler Optimizations

The map-reduce semantics are granularity agnostic, and can be extended all the way to single arithmetic operations. The overhead of the predicate dispatch mechanism and Merge runtime can be overwhelming, however, at finer granularities. Although of limited benefit in absolute performance terms, maintaining the full map-reduce decomposition provides a generic, platform independent descrip-

tion of the algorithm that can target very fine-grain architectures, be translated to other programming models, or be used to drive platform specific optimizing compilers.

For those platforms that would be sensitive to dispatch overhead, the decomposition needs to be cut off earlier, at a coarser granularity. When custom implementations for a specific architecture are available, this cutoff will happen automatically. While it would be possible to create a custom implementation for each kernel on each platform, the combinatorics makes such an undertaking infeasible. When there is sufficient compiler support, automatic library synthesis, based on the source to source translation of the restricted semantics of the map-reduce constructs, can be used to provide kernels at a coarser granularity. Synthesis is performed at the level of single functions, wrapping unit-node variants in loops to create coarse grain function variants, and across functions. For multi-function synthesis, the compiler traces down the hierarchy of map-reduce calls, producing a well structured loop nest suitable for optimizing compilers.

Library synthesis would reduce the number of functions that must be implemented to ensure a sufficiently rich function library. If each kernel in the library, like k-means, includes a full map-reduce version as the default implementation, sequential implementations of varying granularities can be synthesized from the master representation as needed, leaving the developer to focus on more difficult, but higher performing variants using IA32 SSE extensions, the GMA X3000, and other available accelerators.

Function variants are ordered by group and specificity, with ambiguities resolved by a secondary mechanism. As the number of architectures and variants grow, and systems begin to include multiple specialized accelerators, judicious user-driven variant selection will become more difficult. Function variant selection is readily augmented with profiling information, similar to that suggested as an extension in (11). The Merge runtime can maintain relative performance statistics for each applicable variant, generated through a sampling procedure, and biases the variant selection order based on those statistics. Profile guided variant ordering will be particularly important when library synthesis is used, which will create many new variants and has the potential to cutoff selection and invocation before it reached a high-performance implementation buried several layers down. With performance-based ordering, the buried implementation would be selected preferentially despite being more generic than the synthesized counterpart.

4. Evaluation

A prototype of the Merge framework compiler and runtime has been implemented, targeting both heterogeneous and homogeneous multi-core platforms. The key characteristics of the prototype and the results achieved are summarized in the following subsections.

4.1 Prototype Implementation

The Merge source-to-source compiler is implemented as two independent phases: bundling and pattern conversion. Both phases are implemented using OpenC++ (5) and translate C++ with Merge keywords to standard C++ with calls into the Merge runtime. The resulting source is compiled into a single binary by an enhanced version of the Intel C++ Compiler (13) that supports EXOCHI (31). The flow of the Merge framework is shown in Fig. 6.

Function bundling is a global operation performed on classes identified with a bundle keyword. Function variants are bundled based on the method name, with the wrapping class serving as a unique namespace. Each variant can thus be invoked specifically through its fully qualified name (and be generally backward compatible) or through the meta-wrapper. The class wrappers also provide a connection to the existing class hierarchy for group assignment. If not inherited from any class, a particular variant is assumed

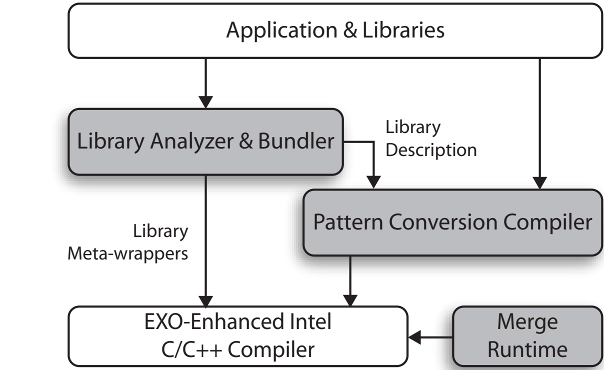

Figure 6. Merge compilation flow.

to belong to the generic group. If the wrapper inherits from a class, the particular variant belongs to the group named by the parent class. At compile time the predicates are converted to logical axioms that can be used with an automated theorem prover CVC3 (3) to determine specificity and exhaustiveness (similar to the methodology and implementation in JPred (22)). In the prototype, dispatch order for ambiguous variants is determined by compile time order, relying on the programmer to order the variants as appropriate.

Pattern conversion is performed on map-reduce statements, replacing the `mapreduce` statement with code implementing the variant selection and invocation described previously. Tasks are created and inserted into the task queue with calls into the Merge runtime. The Merge runtime is built on an enhanced version of the Intel Threading Building Blocks (TBB) (14). TBB provides a generic task driven, fork-join parallel execution runtime for shared memory platforms that has been extended to support additional dependent tasks beyond the spawning parent. A subset of the compiler optimizations described in Section 3.4 are implemented in the Merge framework prototype; both single function library synthesis and limited function variant profiling are currently supported.

4.2 Evaluation Platforms

Heterogeneous: The heterogeneous system is a 2.4 GHz Intel Core 2 Duo processor and an Intel 965G Express Chipset, which contains the integrated Intel Graphics Media Accelerator X3000. The X3000 contains eight programmable, general purpose graphics media accelerator cores, called Execution Units (EU), each of which supports four hardware thread contexts. The details of the EXO implementation for the X3000 is described in (31).

The X3000 ISA is optimized for data- and thread-level parallelism and each EU supports wide operations on up to 16 elements in parallel. The X3000 ISA also features both specialized instructions for media processing and a full complement of control flow mechanisms. The EUs share access to specialized, fixed function hardware. The four thread contexts physically implemented in each EU alternate fetching through fly-weight switch-on-stall multi-threading.

Homogeneous: The homogeneous platform is a Unisys 32-way SMP system using 3.0 GHz Intel Xeon MP processors.

4.3 Performance Analysis

A set of informatics benchmarks has been implemented with map-reduce calls. Table 2 summarizes the benchmarks and inputs. All benchmarks use single precision floating point. For each benchmark, single-threaded straight C reference and Merge implementations were created. When beneficial, component kernels are implemented on the GMA X3000 using hand coded assembly. The porting time represents the entire time needed to create the Merge im-

Table 2. Benchmark Summary

Kernel	Data Size	Description	Porting Time
KMEANS	k=8; 1000000x8 k=8; 10000000x8	k-means clustering on uniformly distributed data	3 days
BLKSCHL	10000000 options 100000000 options	Compute option price using BlackScholes algorithm	4 days
LINEAR	640x768 image 2000x2000 image	Linear image filter – compute output pixel as average of 9 pixel square	2.25 days
SVM	480x704 image 720x1008 image	Kernel from SVM-based face classifier	2 days
RTSS	64x3600000 samples	Euclidean distance based classification component of real-time spike sorting algorithm used in neural prosthetics (32)	1.5 days
BINOMIAL	10000 options	Compute options price using binomial algorithm for 500 time steps	5 days
SEPIA	640x768 image 2000x2000 image	Modify RGB values to artificially age image	1.75 days
SMITHWAT	800 base pairs 1000 base pairs	Compute scoring matrix for pair of DNA sequences using SmithWaterman algorithm	.5 days

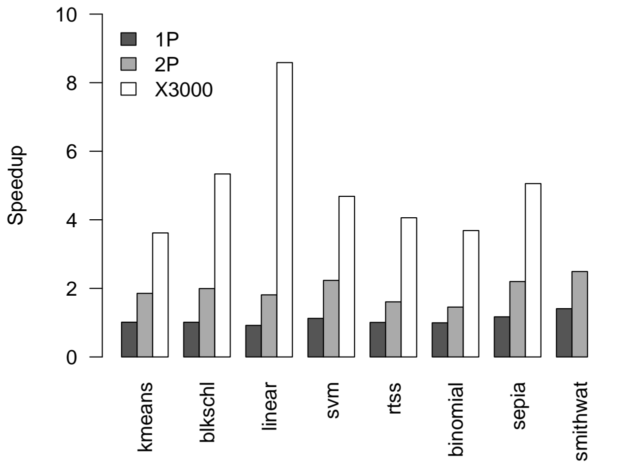

Figure 7. Speedup for kernels using Merge framework on 1 or 2 IA32 cores and 2 IA32 cores plus the Intel GMA X3000 vs. straight C reference implementation on a single IA32 core

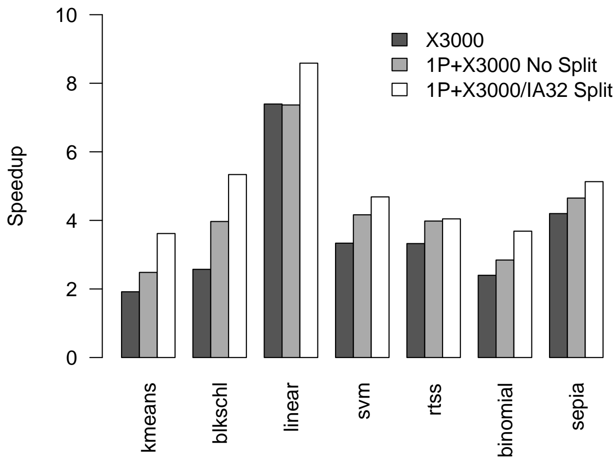

Figure 8. Speedup for kernels using Merge framework on heterogeneous system using different cooperative multi-threading approaches between the Intel GMA X3000 and IA32 CPU

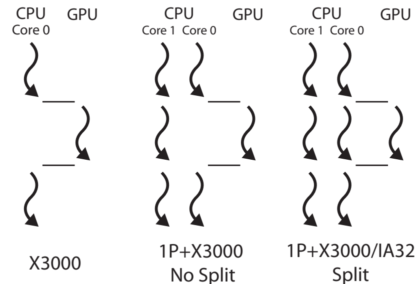

Figure 9. Sketch of thread execution under different cooperative multi-threading approaches

plementation, including the creation of variants for the X3000. Developing the X3000 variants is responsible for most of the porting time, best exemplified by **RTSS** which was relatively quick to port because it reused some X3000 variants developed for **KMEANS**.

All benchmarks are compiled with an enhanced version of the Intel C++ Compiler using processor specific optimization settings (/fast). These compiler optimizations include auto-vectorization and tuning specifically for the Intel Core 2 Duo and Intel Xeon processors in the test platforms. Performance is measured using wall clock time with the timing facilities provided by the TBB runtime.

Fig. 7 shows the speedup achieved, relative to the straight C reference implementation, for the benchmarks using the Merge framework for one and two IA32 processor cores and the Intel GMA X3000. The Merge framework implementation shows performance comparable to straight C reference implementation on one processor core (.9x or better) and achieves meaningful speedup (1.45x-2.5x) executing on two processor cores. When X3000 variants are available, additional speedup (3.7x-8.5x relative to reference implementation) is obtained, exceeding what was achieved on the dual core processor alone. The parallelization for multiple cores and utilization of the X3000 are automatic and do not require any changes to the source. **SMITHWAT** utilizes a dynamic programming algorithm and is not well suited to X3000 so no variants were developed for that architecture; it is included as an example of the indirect communication capabilities of the Merge framework parallel execution model and will be discussed in more detail below.

The GMA X3000 is highly optimized for image and video processing applications, and as expected shows the best performance on the purely image processing benchmarks, particularly **LINEAR** and **SEPIA**. The other benchmarks are much less image oriented,

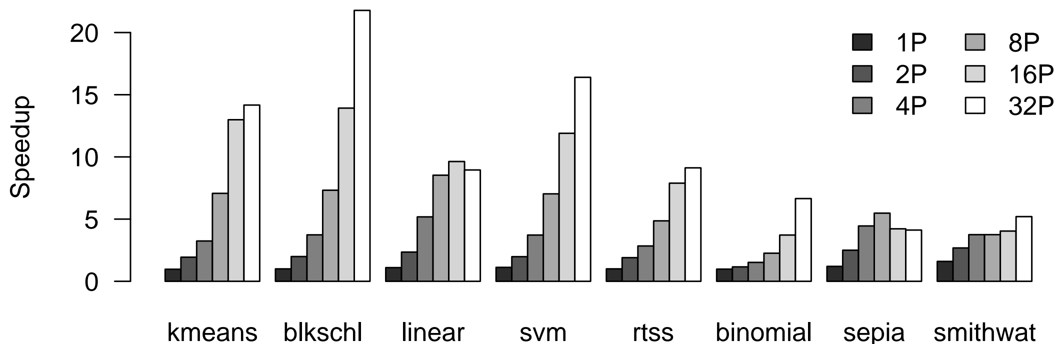

Figure 10. Speedup for kernels using Merge framework on the homogeneous 32-way SMP machine vs. straight C reference implementation on a single core

with lower computation/bandwidth ratios, and thus do not achieve the same level of speedup on the X3000. Benchmarks with similar CPU and X3000 performance can often benefit from cooperative multi-threading, in which the X3000 and IA32 CPU concurrently process disjoint subsets of the map operations. In these cases the X3000 functions less as a distinct accelerator and more as an additional processor core, tightly coupled to the traditional CPU(s).

Fig. 8 shows the speedup achieved using different cooperative multi-threading strategies. The activity of the processors under the different strategies is sketched in Fig. 9. When the accelerator code is invoked using EXOCHI, the CPU thread manages the accelerator, waiting until all of the accelerator threads have completed before proceeding. Thus if only one CPU core is used (X3000), the CPU and X3000 do not execute concurrently, and any and all speedup results entirely from using the X3000. This is the level of performance that might be expected on system that focuses only on compiling general purpose code to specialized accelerators.

When multiple CPU cores are used (1P+X3000/1P No Split), the Merge framework automatically distributes work between the accelerator and the second CPU core. In this regime, the runtime first attempts to invoke the X3000 variant, if the X3000 is busy, the runtime defaults to the next best variant, which in this case targets the CPU. For the less image oriented benchmarks, such as **KMEANS** and **BLKSCHLS**, where the CPU performs well relative to the X3000, the automatic cooperative multi-threading can significantly improve performance. A 28% and 54% performance increase over X3000 was obtained for **KMEANS** and **BLKSCHLS** respectively.

In the first two regimes, the CPU thread which serves as the manager for the accelerator threads idles until the accelerator completes. EXOCHI supports limited nowait fork-join, in which the CPU master thread can continue to perform useful work before blocking at the join point. Using the nowait fork-join, the working set for the variant can be partitioned between the CPU and GPU (1P+X3000/IA32 Split), in effect creating joint CPU-GPU variant. The work distribution is static, and is selected so the CPU and X3000 finish as close to the same time as possible. In this regime, which makes use of all the CPU and accelerator cores, additional improvements of 45% and 34% were obtained over the 1P+X3000/1P No Split regime for **KMEANS** and **BLKSCHLS**. Extending the EXOCHI and Merge prototypes to support full dynamic scheduling of the manger CPU thread is an area of ongoing work. Note that this static partitioning does not affect the dynamic scheduling of the other cores or accelerators.

Fig. 10 shows the speedup relative to the straight C reference implementation achieved on the homogeneous test platform for the same benchmarks (with the larger input datasets). As with the heterogeneous platform, using the 32-way SMP system did not require any changes to the source.

The CPU-only benchmark **SMITHWAT** takes advantage of the indirect communication provided by the Merge framework parallel execution model. The benchmark produces the scoring matrix used for biological sequence alignment with a dynamic programming (DP) algorithm. The DP algorithm creates dataflow dependencies between the otherwise independent entries in the score matrix, which in a parallel model that only supports direct communication through task call and return is cumbersome to express. Using indirect communication via the collection arguments, the task for a matrix entry can register its dependencies and switch-out until the required entries have been computed, allowing the Smith-Waterman algorithm to be effectively parallelized without resorting to highly specialized implementations or DP-specific parallel patterns.

When highly optimized parallel implementations are available for an algorithm, such as for IA32 SSE, or using existing low-level thread constructs, they can be incorporated into the library and invoked directly. Thus in general, the performance of the Merge framework's library-based approach will track that of the best available implementations, ensuring that even in the worst case the Merge framework implementation will perform comparably to existing implementations.

Note that the X3000 used in the heterogeneous test platform is a chipset integrated GPU and does not have the computational resources or memory bandwidth of a discrete GPU, which limits performance. However as the real speedups shown here indicate, the X3000 can provide useful acceleration, especially when used with the CPU as part of a whole system approach. Further, many users will already have an X3000 or similar GPU available in their systems, offering additional speedup relative to the CPU alone for no additional hardware cost. And as described earlier, the Merge framework can be used with any accelerator that can support the EXO interface, including discrete GPUs.

5. Related Work

Programming models for heterogeneous systems can be broadly categorized into direct and library-based approaches. Direct approaches, such as the DSVMs cited previously or streaming languages (4; 30; 15; 20; 24), map the application directly to the accelerator ISA or API, while library-based approaches, such as the Merge framework, Sequoia, and MATCH (23), map the application to a library of function-intrinsics that encapsulate the accelerator-specific code. The key difference between the two models lies in their ability to abstract and exploit features of the underlying accelerator; direct approaches are limited to the features that can be abstracted by their language primitives, while library-based ap-

proaches can abstract any feature. The tradeoff is that the library-based compiler will have only minimal information about any given function-intrinsic, and thus will miss some of the optimizations exploited by the more restrictive direct approach compilers.

Although the direct approaches treat the accelerator as a co-processor for the general purpose CPU, these programming models primarily focus on supporting general purpose computation on specialized hardware. The accelerator focus is only magnified for device-specific languages, such as GPU shader languages: Cg (19), OpenGL Shader Language and HLSL. Device specific languages or APIs, such as the shader languages, ATI's Close-to-Metal (28), OpenGL and DirectX, could be building blocks, however, for GPU-specific function implementations that are then integrated with the Merge framework for use alongside implementations targeting other accelerator architectures.

The library-based compilation of the Merge framework is inspired by: the static MATCH library-based compiler for Matlab; library meta-programming approaches, specifically the annotation and selection mechanisms (11; 16); the traits-based multiple dispatch in the Fortress language (1); and the static task-based function variant selection in Sequoia. The Merge framework similarly attempts to improve performance by optimizing across function calls using programmer-supplied annotations. However, the above approaches choose implementations at compile time, either requiring all selection parameters (*e.g.* array sizes) to be static, or forcing runtime specialization, created with `if-else` statements, into the function implementations, making applications non-portable and difficult to maintain. The additional overhead of runtime specialization in the Merge framework is mitigated by assembling, either manually or through library synthesis, sufficiently coarse-grain functions.

The Intel C++ Compiler (13) automatically generates and transparently selects among multiple variants of compiled code, specific to certain architectural features like SSE. The Merge framework incorporates the machine architecture as a parameter in the function selection system, providing similar, automatic, architecture-specific dispatch, but for the broader set of accelerators that can support an EXOCHI interface (including those without significant compiler support). Since selection among target variants is automatic, the user does not need to manually specify the variants, which requires an intimate knowledge of the available processors and the function library.

The Merge framework's task-driven execution model and parallel programming language is based on the map-reduce pattern that is present in many functional languages and underlies Google's MapReduce programming model; Phoenix (27) (a shared memory implementation of MapReduce); the similar constructs in Sequoia, and generator-reducer driven parallelism in Fortress. The Merge framework is not just an implementation of Google's MapReduce library, however. As in Sequoia, the map and reduce constructs in the Merge framework are used as an efficient set of semantics for describing the potential concurrency in an algorithm.

6. Conclusion

The Merge framework has the potential to replace current ad hoc approaches to parallel programming on heterogeneous systems with a rigorous, library-based methodology. Rather than writing or compiling the application to specific accelerator targets, the programmer expresses computations using architecture-independent, high-level language extensions based on the map-reduce pattern. Merge automatically and dynamically parallelizes, maps and load-balances the computation over the heterogeneous cores available in the computer system. Computation is mapped to a set of function-intrinsics, which encapsulate the accelerator-specific code. The Merge framework is applicable to many heterogeneous systems, as the function-intrinsics can abstract computations of any complexity, and for any architecture that supports a lightweight sequencer interface. Merge-based applications are easily extensible, and can readily target new accelerator architectures, or different system configurations. Using the Merge programming interface, we demonstrate significant speedup for several applications using the same source code on both a heterogeneous system consisting of a dual core CPU and an integrated 8-core, 32-thread GPU and homogeneous 32-way SMP system.

Acknowledgments

We would like to thank Perry Wang, Hong Jiang, Xinmin Tian, and Ghassan Yacoub for their help with this project. We also appreciate the support of Shekhar Borkar and Joe Schutz. In addition, we would like to thank Ethan Schuchman, Anne Bracy, James Balfour, Omid Azizi, and the anonymous reviewers whose valuable feedback has helped the authors greatly improve the quality of this paper. This work was partially supported by the Focus Center for Circuit and System Solutions (C2S2), one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation Program.

References

[1] E. Allen, D. Chase, J. Hallet, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt. The Fortress language specification version 1.0beta. Technical report, Sun Microsystems, 2007.

[2] M. Annavram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *Proc. of ISCA*, pages 298–309, 2005.

[3] C. Barret and S. Berezin. CVC lite: A new implementation of cooperating validity checker. In *Proc. of Conf. on Computer Aided Verification*, pages 515–518, 2004.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[5] S. Chiba. A metaobject protocol for C++. In *Proc. of OOPSLA*, pages 285–299, 1995.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI*, pages 137–149, 2004.

[7] P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Technology@Intel Magazine*, 2005.

[8] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *European Conf. on Object-Oriented Programming*, pages 186–211, 1998.

[9] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequioa: Programming the memory hierarchy. In *Proc. of ACM/IEEE Conf. on Supercomputing*, page 83, 2006.

[10] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of both latency and throughput. In *Proc. of ICCD*, pages 236–243, 2004.

[11] S. Z. Guyer and C. Lin. Annotation language for optimizing software libraries. In *Proc. of Conf. on Domain Specific Languages*, pages 39–52, 1999.

[12] R. Hankins, G. Chinya, J. D. Collins, P. Wang, R. Rakvic, H. Wang, and J. Shen. Multiple instruction stream processor. In *Proc. of ISCA*, pages 114–127, 2006.

[13] Intel. Intel C++ compiler. http://www3.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm.

[14] Intel. Intel threading building blocks. http://www3.intel.com/cd/software/products/asmo-na/eng/294797.htm.

[15] U. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. Owens. Programmable stream processors. *IEEE Computer*, 36(8):54–62, 2003.

[16] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proc. of the IEEE*, 93:378–408, 2005.

[17] V. Kuncak, P. Lam, and M. Rinard. Role analysis. *ACM SIGPLAN Notices*, 37:17–32, 2002.

[18] P. Lyman and H. R. Varian. How much information. http://www.sims.berkeley.edu/how-much-info-2003, 2003.

[19] W. Mark, R. Glanville, K. Akeley, and M. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[20] M. McCool and S. Toit. *Metaprogramming GPUs with Sh*. A K Peters, 2004.

[21] M. McCool, K. Wadleigh, B. Henderson, and H. Y. Lin. Performance evaluation of GPUs using the RapidMind development platform. In *Proc. of ACM/IEEE Conf. on Supercomputing*, page 81, 2006.

[22] T. Millstein. Practical predicate dispatch. In *Proc. of OOPSLA*, pages 345–264, 2004.

[23] A. Nayak, M. Haldar, A. Kanhere, P. Joisha, N. Shenoy, A. Choudhary, and P. Banerjee. A library based compiler to execute MATLAB programs on a heterogeneous platform. In *Proc. of Conf. on Parallel and Distributed Computing Systems*, 2000.

[24] NVidia. Cuda. http://developer.nvidia.com/object/cuda.html.

[25] Peakstream. The PeakStream platform: High productivity software development for multi-core processors. Technical report, PeakStream Inc., 2006.

[26] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *Proc. of ISSCC*, pages 184–185, 2005.

[27] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. of HPCA*, pages 13–24, 2007.

[28] M. Segal and M. Peercy. A performance-oriented data parallel virtual machines for GPUs. Technical report, ATI Technologies, 2006.

[29] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In *Proc. of ASPLOS*, pages 325–335, 2006.

[30] W. Thies, M. Karczmarek, and S. Amarainghe. StreamIt: A language for streaming applications. In *Proc. of Conf. on Compiler Construction*, pages 49–84, 2002.

[31] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proc. of PLDI*, pages 156–166, 2007.

[32] Z. S. Zumsteg, C. Kemere, S. O'Driscoll, G. Santhanam, R. E. Ahmed, K. V. Shenoy, and T. H. Meng. Power feasibility of implantable digital spike sorting circuits for neural prosthetic systems. *IEEE Trans Neural Syst Rehabil Eng*, 13(3):272–279, 2005.