Embracing Heterogeneity — Parallel Programming for Changing Hardware

Michael D. Linderman, James Balfour, Teresa H. Meng and William J. Dally
Center for Integrated Systems and Computer Systems Laboratory, Stanford University
{mlinderm, jbalfour}@stanford.edu

Abstract

Computer systems are undergoing significant change: to improve performance and efficiency, architects are exposing more microarchitectural details directly to programmers. Software that exploits specialized accelerators, such as GPUs, and specialized processor features, such as software-controlled memory, exposes limitations in existing compiler and OS infrastructure. In this paper we propose a pragmatic approach, motivated by our experience with Merge [3], for building applications that will tolerate changing hardware. Our approach allows programmers to leverage different processor-specific or domain-specific toolchains to create software modules specialized for different hardware configurations, and it provides language mechanisms to enable the automatic mapping of the application to these processor-specific modules. We show this approach can be used to manage computing resources in complex heterogeneous processors and to enable aggressive compiler optimizations.

1 Introduction

Heterogeneous computer systems, which may integrate GPUs, FPGAs and other accelerators alongside conventional CPUs, offer significantly better performance and efficiency. However, they often do so by exposing to programmers architectural mechanisms, such as low-latency scratchpad memories and inter-processor interconnect, that are either hidden or unavailable in general-purpose CPUs. The software that executes on these accelerators often bears little resemblance to its CPU counterpart: source languages and assembly differ, and often entirely different algorithms are needed to exploit the capabilities of the different hardware.

The ISAs of commodity general-purpose processors have changed remarkably little during the past 30 years. Decades old software still runs correctly, and fast, on modern processors. Unlike their ISAs, processor microarchitectures and system architectures have changed significantly. As modern architectures expose more microarchitectural and system details to software to improve performance and efficiency, programmers are no longer insulated from the evolution of the underlying hardware. Programming models need to be inclusive of different processor architectures, and tolerant of continual, often radical, changes in hardware.

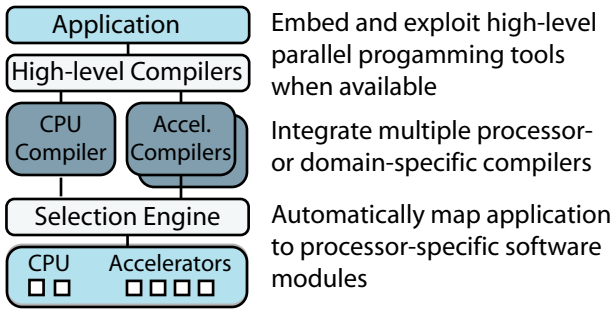

Figure 1: Sketch of Merge framework

To exploit these new and different hardware resources, a diverse set of vendor-specific, architecture-specific and application-specific programming models have and are currently being developed. The rapid evolution of hardware ensures that programming models will continue to be developed at a torrid pace. Integrating different toolchains, whether from different vendors or using different high-level semantics, remains a challenge. However, integrating many narrowly-focused tools is more effective than attempting to craft a single all-encompassing solution; consequently, that is the approach we take.

In this paper, we present a methodology, motivated by our experiences with the Merge framework [3], for building programs that target diverse and evolving heterogeneous multicore systems. Our approach, summarized in Figure 1, automatically maps applications to specialized software modules, implemented with different processor-specific or domain-specific toolchains. Specialized domain-specific languages and accelerator-

specific assembly are *encapsulated* in C/C++ functions to provide a uniform interface and inclusive abstraction for computations of any complexity. Different implementations of a function are *bundled* together, creating a layer of indirection between the caller and the implementation that facilitates the mapping between application and implementation.

Section 2 motivates the use of encapsulation and bundling, summarizing and updating the techniques first described in [3]. Sections 3 and 4 present our most recent work, in which we show how encapsulation and bundling can be used to effectively manage computing resources in complex heterogeneous systems and enable aggressive compiler optimizations.

2 An Extensible Programming Model

The canonical compiler reduces a computation expressed in some high-level language to a small, fixed set of primitive operations that abstract the capabilities of the target hardware. Compilation and optimization strategies are biased by the choice of primitive operations. Optimizations developed for one set of primitives are often of limited use when the primitive operations fail to abstract important aspects of the target hardware or application.

Unfortunately, no one set of primitive operations can effectively abstract all of the unique and specialized capabilities provided by modern hardware. For instance, the capabilities of scalar processors are represented well by three-address operations on scalar operands; the capabilities of SIMD processors, such as Cell, GPUs and SSE units, are better represented by short-vector operations; and the capabilities of FPGAs are better represented by binary decision diagrams and data flow graphs with variable-precision operands. Much as the limitations of scalar primitives motivated the adoption of short-vector primitives in compilers targeting SIMD architectures, compilers that target complex accelerators such as FPGAs will find representations based on simple scalar and short-vector primitives limiting and ineffective.

We argue that nascent parallel programming systems should allow software that uses different programming models and primitives to be integrated simply and efficiently. These systems require *variable* and *inclusive* primitives, primitives that can abstract computational features of any complexity (variable) and for any architecture, or using any programming model (inclusive).

2.1 Encapsulating Specialized Code

Fortunately, programming languages already provide variable and inclusive primitives: functions. Programming systems such as EXOCHI [7] and CUDA [5] allow programmers to inline domain-specific languages

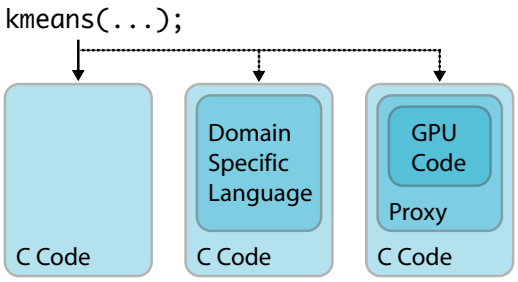

Figure 2: Encapsulation of inline accelerator-specific assembly or domain-specific languages

(DSLs) and accelerator-specific assembly into C-like functions, thereby creating a uniform interface, compatible with existing software infrastructure, that is independent of the actual implementation. Figure 2 shows an example in which **`kmeans`** is implemented using combinations of standard C, a DSL, and GPU-specific assembly. All versions present the same interface and all appear to the caller to execute in the CPU memory space. The proxy layer (e.g., EXOCHI, CUDA) provides the data transfer and other runtime infrastructure needed to support the interaction between the CPU and the accelerator.

These enhanced functions, which we term *function-intrinsics*, are conceptually similar to existing compiler intrinsics, such as those used to represent SSE operations. Unlike conventional intrinsics, programmers are not limited to a small fixed set of operations; instead, programmers can create intrinsics for operations of any complexity, for any architecture and using any programming model supported by a proxy interface. When programmers use a non-C language, such as GPU assembly, the appropriate compiler is invoked and the resulting machine code (or an intermediate language and a just-in-time compiler or interpreter) is packaged into the binary.

2.2 A Concurrent Function Call ABI

Using the function call interface to integrate specialized implementations is actually common. For example, most systems ship with a version of the C standard library that is optimized for that particular platform. Often the optimized implementation includes machine-specific assembly and operating system specific system calls. We extend this approach to more than just a few standardized libraries. We believe programmers will need to extend and specialize many different APIs to exploit different hardware efficiently.

The simple and complete definition of the C function call ABI provides a reasonable starting point, but must be enhanced to provide guarantees needed for correct concurrent execution. Additional restrictions are required to ensure different implementations of the same function can be invoked interchangeably, independently

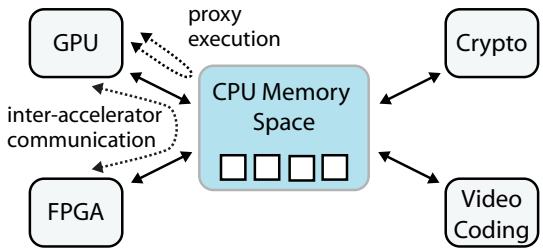

Figure 3: Relationship between different accelerators and the CPU, which acts as a hub.

and potentially concurrently. Thus, we require that all function-intrinsics be independent and potentially concurrent; only access data passed as arguments; execute atomically with regards to each other; and limit direct communication to call and return operations.

From the perspective of the function caller on the CPU, the computation starts and completes on the CPU, and all communication occurs through the CPU. Accordingly, we make the CPU and its memory space of the hub of the system (Figure 3). This organization reflects the typical construction of computer systems, in which the CPU coordinates activities throughout the system.

2.3 Bundling Function Intrinsics

Since any one implementation of a function may not be the most efficient for all inputs, multiple implementations should be allowed to coexist. Dynamically selecting which implementation to use allows an application to perform well across different workloads and different platforms. Conventional systems use combinations of static and dynamic techniques (**`#ifdef`**, dynamic/static linking, **`if-else`** blocks) to select implementations. For example, the C standard library is specialized through system-specific linking. However, as the diversity of heterogeneous systems increases and systems with multiple accelerators become commonplace, the number of available implementations will make such approaches impractical. The problem is particularly acute if programmers must manually select implementations.

We replace the current ad-hoc use of static and dynamic selection process with a unified approach built around predicate dispatch [4, 6]. Predicate dispatch subsumes single and multiple dispatch, conditioning invocation on a boolean predicate function that may include the argument types, values, and system configuration. A programmer supplies a set of annotations with each function implementation. These annotations provide a common mechanism for describing invariants for a given function, and are independent of the programming model used to implement the particular function intrinsic.

There are three classes of annotations: *input restrictions*, which are boolean restrictions on the input arguments (e.g. data set size < 10000); *configuration restrictions*, which specify the necessary compute resources (e.g. availability of a suitable GPU); and *traits*, which describe properties that are useful to users of the function (e.g. associativity). At compile time, when function variants implementing the same computation are bundled together, the annotations are analyzed and translated into a set of dispatch wrapper functions that implement the generic function interface and provide introspection into the variants available in the bundle.

The dispatch wrappers can be used to automatically select an implementation, freeing the programmer from having to manually map an application to particular function-intrinsics. A particular variant is selected by evaluating the annotations for each function-intrinsic until a variant whose annotation predicates evaluate to **`true`** is found. In addition to ensuring that only applicable function variants are invoked, the dispatch wrappers provide basic load balancing. The dispatch system checks the dynamic availability of the requested resources before invoking a variant. Thus, it will not, for example, invoke a function-intrinsic that targets the GPU if the GPU is being used by the graphics subsystem. Variants are ordered by annotation specificity, performance, and programmer-supplied hints [3].

In its simplest use, the dispatch system transparently selects a particular function intrinsic. The objective function used in the scheduling algorithm, greedy selection based on the ordering described above, is implicit in the implementation of the meta wrappers. The tradeoff is that the compiler and runtime must infer a "good" objective function for a particular application and set of machine configurations. However, the results presented in [3] show that good performance can be achieved using these very simple inferred objective functions. For those programmers and programs that require more control, alternate scheduling approaches could be used. One such example is described below. However, by making automatic and transparent selection the default, non-expert programmers are not obligated to immerse themselves in the details of the particular specialized function-intrinsics that might be available.

In more advanced usage, the programmer might explicitly uses the introspection capabilities offered by the dispatch wrappers to implement additional functionality, such as more sophisticated schedulers, on top of the core bundling infrastructure. For example, the Harmony programming model [1] implements first-to-finish scheduling of different kernels onto heterogeneous compute resources. Presently, at scheduling time, the Harmony runtime computes the intersection between the implementations available and the installed processors to determine the set of kernels over which the computation can be scheduled. Using the Merge bundle system, a system

like Harmony could provide more comprehensive specialization. Instead of a single predicate, processor architecture, the intersection can include an arbitrary set of conditions on the input or machine configuration. In this usage model, the Harmony-like runtime would explicitly query the function bundles for all applicable implementations, and then choose among based on its own scheduling algorithm.

3 Managing Resource Sets

Extensive resource virtualization in commodity general-purpose processors has allowed programmers to largely ignore resource management. However, the hardware required for virtualization, such as TLBs, is expensive and rarely implemented in accelerators, such as GPUs. For example, CUDA programmers must explicitly manage the GPU's scratchpad memory. For the same efficiency reasons, embedded systems often do not virtualize hardware resources; programmers must explicitly allocate resources, such as memory and bandwidth, in modern heterogeneous SoCs. However, for a number of embedded applications, notably cell phones, market pressures favor opening systems to third-party application programmers, bringing issues of resource protection and allocation to the forefront.

General-purpose processor-like virtualization is ineffective for heterogeneous systems. In the current model, the programmer can only control a virtualized time-slice on a single core, which is insufficient for managing small software controlled memories or bandwidth to a shared resource, such as a crypto accelerator. To efficiently exploit diverse hardware resources, programmers need to be able to assemble more complex resource sets. For example, two processors, shown in Figure 4, that share a dedicated communication link and can be scheduled and managed as a single resource. However, allocation cannot be all-or-none if resources are to be shared among multiple clients. For example, allocating an entire accelerator, such as the GPU, to a single process is wasteful if the process cannot fully utilize it. Flexible resource sets, a compromise between current general-purpose and embedded approaches, can address this problem.

Flexible resource sets allow programmers to assemble multiple, otherwise independent resources into a single unit when needed. We can consider each resource set to be a unique hardware resource, and sometimes even a different class of processor that might favor a different programming model. For example, tiled architectures might be treated as many independent tiles and programmed using existing threading frameworks (e.g., POSIX threads), or might be treated as a single coordinated systolic array and programmed using a streaming language [2]. And between these extremes, there are

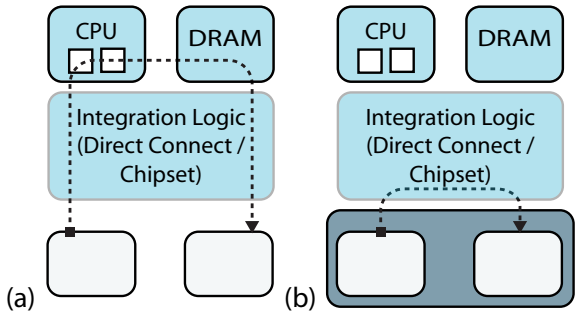

Figure 4: Combining accelerators (a) to create new resource sets to achieve performance guarantees, or exploit dedicated communication resources (b).

usage models that blend the high-level streaming language with custom-implemented kernels that use low-level threading primitives. The flexible encapsulation, annotations and function overloading provides the necessary compiler infrastructure to support flexible resource sets.

Different programming models, possibly targeting different resource sets, can be encapsulated in C/C++ functions. The proxy layer, shown in Figure 2, allows resources that an OS normally considers independent to be grouped into a single OS resource in which most of the resources are explicitly managed by the programmer [8]. For example, n cores appear to the OS as one, with system calls for the $n-1$ cores proxied through the one exposed to the OS. The **`configuration`** annotations allows programmers to tell the compiler and runtime what resources are required for each function intrinsic.

Without virtualization, resource allocations requests are more likely to fail. Applications must include tedious and error-prone boiler-plate code to test the availability, allocate and recover from the failure to allocate, heterogeneous resources. Predicate dispatch, controlled by the configuration annotations, replaces the current ad-hoc approach to resource management. The compiler translates the annotations into calls into the appropriate driver to query availability and allocate resources. If any part of the request fails, the runtime can automatically invoke alternate implementations provided in the function bundles. New or different fallback implementations can be integrated as new function-intrinsics; no changes to existing code, such as adding **`if-else`** statements to explicitly control fallback on failure, are required.

Flexible resource sets will be hidden from most programmers behind library APIs. For those programmers that need more control, the Merge approach provides a framework for integrating implementations that target more specific sets of resources. By collecting otherwise independent resources together to create units that are allocated and scheduled as a single resource, systems can preserve the conventional CPU-centric archi-

```
matrix H(matrix A, matrix B, matrix C) {
  matrix T1 = F(A, B);
  matrix T2 = G(T1, C);
  return T2;
}
```

Figure 5: Example function that could benefit from inter-procedural optimization

tecture shown in Figure 3 and leverage existing software infrastructure, such as the OS, while exploiting inter-accelerator interconnect and other difficult to virtualize resources. The combination of configuration annotations and runtime function variant selection provides a limited form of OS-like resource protection and allocation until more sophisticated OS-infrastructure is developed.

4 Compiler Optimizations

Successfully exploiting complex heterogeneous systems requires the programmer assemble appropriate resource sets (described in Section 3) and smartly structure the computation to take advantage of those resources. For example, to profit from offloading a computation to a discrete GPU, the computation must have enough arithmetic intensity to amortize the latency of transferring data between the CPU and GPU. Identifying an appropriate granularity at which to offload computation to specialized accelerator is one of the key challenges of heterogeneous systems.

Consider the pseudo-code in Figure 5, in which two functions are called in sequence. In the simplest use of Merge, the `F` and `G` function calls could be independently mapped to different hardware resources, with data copied between the CPU and accelerator memory spaces as needed. For some inputs, the overhead of the data copying will be adequately amortized, and this approach will be satisfactory. Dispatch annotations, supplied by the programmer or generated through execution-profiling, can be used to limit the invocation of a particular implementation to just those inputs for which it will be beneficial. The `H` function is no different; it can also be mapped to different implementations. If programmers desire better performance, they can create a new optimized implementation of `H`, in effect inter-procedurally optimizing across `F` and `G`, that can be bundled alongside the version in Figure 5.

When there is little or no sophisticated compiler support, there is no other option than for the programmer to manually build up optimized implementations. Many of the function-intrinsics developed for the Intel X3000 integrated GPU in [3] were implemented this way. Functions were fused together until the function-intrinsic performed enough computation to amortize the data transfer latency. As compiler support improves these optimizations will be automated. The encapsulation and bundling in Merge can facilitate these inter-procedural optimizations. Encapsulated languages provide the input for the optimization, with the product, and its associated dispatch annotations, integrated into the function bundles as an alternate implementation.

The compiler can implement effective optimizations with only a basic understanding of the target architecture. For example, for sequentially invoked functions, like `F` and `G` in Figure 5, we are developing tools to eliminate intermediate data transfers on CUDA-enabled GPUs. The optimizer queries the `F` and `G` function bundles for CUDA implementations. If they are found, the optimizer creates a new implementation of `H` in which `F` and `G` are inlined and the intermediate copies eliminated. This tool does not need to understand the GPU-code, it just needs to be able to identify data transfers and inline calls to CUDA device functions (similar to inlining C++ function calls).

With a deeper understanding of the target architecture more sophisticated optimizations are possible. However, specialized implementations, such as those written in assembly or a low-level language and intended for direct execution on particular processor, are rarely a good starting point for optimization. In these cases, we can exploit the encapsulation and bundling capabilities to integrate implementations using high-level DSLs, such as streaming languages, that better support aggressive optimizations. These encapsulated DSLs are particularly useful for established multicore systems that have sophisticated compiler support, but are nonetheless challenging to program using only low-level tools.

The Merge framework includes a DSL, based on the map-reduce pattern, that provides an expressive and flexible way for programmers to expose parallelism. However, executing unoptimized map-reduce code can impose a significant performance penalty; directly executing the map-reduce implementation of the k-means clustering algorithm on a single core is 5× slower than the C reference implementation. The compiler support for the map-reduce DSL presented in [3] was limited to simple intra-procedural optimizations, and as result, the map-reduce function-intrinsics were primarily used for coarse-grain task-level parallelism (distributed across heterogeneous processors). We are currently developing more advanced, inter-procedural optimizers, targeting x86 processors with SSE extensions and CUDA-enabled GPUs. Preliminary results for the most aggressive optimizations, including inlining, algebraic simplification and automatic vectorization using SSE extensions, show a 1.56× speedup of k-means relative to the C reference implementation on a single processor core.

Optimization of encapsulated DSLs is most useful in the broad middle ground between traditional uniproces-

sors and bleeding-edge accelerators. Uniprocessor systems are readily targeted using conventional programming tools, while new accelerators invariably lack sophisticated compiler support and must be programmed using accelerator-specific assembly or other low-level tools. Function bundling enables implementations targeting both systems to coexist; applications can exploit the newest and most powerful computing resources without compromising performance on legacy architectures. By also including DSL-based function-intrinsics, programmers can leverage steadily improving compiler technology to improve productivity and application performance for established heterogeneous systems. Functions written in the map-reduce DSL, for instance, now benefit from support for SIMD extensions, with support for GPUs forthcoming.

The product of an optimizer, a new function-intrinsic, will just be one of possibly several different implementations for a computation. An optimizer does not need to generate the one best implementation for all scenarios. Instead it can focus of generating a great implementation for a particular input or hardware configuration. Tasks that would be common to many optimizers, such as eliminating unneeded implementations and performance ranking function variants, are provided as part of the bundling infrastructure using static analysis, heuristics and profiling [3]. With a focused mission and powerful supporting infrastructure, optimizers are simpler and easier to build, accelerating the development of sophisticated compiler support for new and evolving hardware.

5 Conclusion

Computer systems will change in significantly the coming decade and beyond. Although steadily improving compiler technology will enable programmers to target more and more different architectures using the same high-level source code, there will always be important accelerators with little or no sophisticated compiler support that require expert-created low-level modules. Enabling the easy integration of different programming models and different processors, and the efficient reuse of expert-developed code will be key to navigating this ongoing transition. In this paper we have presented a pragmatic approach to developing applications for complex heterogeneous systems. We described how function encapsulation and bundling can be used to integrate many different processors, or combination of processors, while also supporting advanced optimization techniques; ensuring that programmers can take advantage of state-of-the-art hardware and compilers tools, as both become available.

6 Acknowledgments

Merge originated during an internship at the Intel Microarchitecture Research Lab and we would like to thank Hong Wang, Jamison Collins, Perry Wang and many others at Intel for their support and help. Additionally we would like to thank Shih-wei Liao, David Sheffield, Mattan Erez and the anonymous reviewers who 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 Systems Solutions (C2S2), one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation Program and the Cadence Design Systems Stanford Graduate Fellowship.

References

[1] DIAMOS, G., AND YALAMANCHILI, S. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proc. of HPDC* (2008), pp. 197–200.

[2] GORDON, M. I., THIES, W., KARCZMAREK, M., LIN, J., MELI, A. S., LAMB, A. A., LEGER, C., WONG, J., HOFFMANN, H., MAZE, D., AND AMARASINGHE, S. A stream compiler for communication-exposed architectures. In *Proc. of ASPLOS* (2002), pp. 291–303.

[3] LINDERMAN, M. D., COLLINS, J. D., WANG, H., AND MENG, T. H. Merge: A programming model for heterogeneous multi-core systems. In *Proc. of ASPLOS* (2008), pp. 287–296.

[4] MILLSTEIN, T. Practical predicate dispatch. In *Proc. of OOPSLA* (2004), pp. 345–264.

[5] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.0 ed., 2008.

[6] PARDYAK, P., AND BERSHAD, B. Dynamic binding for an extensible system. In *Proc. of OSDI* (1996), pp. 201–212.

[7] WANG, P. H., COLLINS, J. D., CHINYA, G. N., JIANG, H., TIAN, X., GIRKAR, M., YANG, N. Y., LUEH, G.-Y., AND WANG, H. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proc. of PLDI* (2007), pp. 156–166.

[8] WANG, P. H., COLLINS, J. D., CHINYA, G. N., LINT, B., MALLICK, A., YAMADA, K., AND WANG, H. Sequencer virtualization. In *Proc. of ICS* (2007), pp. 148–157.