

Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems

Gregory Diamos and Sudhakar Yalamanchili
School of Electrical and Computer Engineering, Georgia Institute of Technology
Atlanta, Georgia, USA
{Gregory.Diamos@gatech.edu, sudha@ece.gatech.edu}

ABSTRACT

The emergence of heterogeneous many core architectures presents a unique opportunity for delivering order of magnitude performance increases to high performance applications by matching certain classes of algorithms to specifically tailored architectures. Their ubiquitous adoption, however, has been limited by a lack of programming models and management frameworks designed to reduce the high degree of complexity of software development intrinsic to heterogeneous architectures. This paper proposes Harmony, a runtime supported programming and execution model that provides: (1) semantics for simplifying parallelism management, (2) dynamic scheduling of compute intensive kernels to heterogeneous processor resources, and (3) online monitoring driven performance optimization for heterogeneous many core systems. We are particularly concerned with simplifying development and ensuring binary portability and scalability across system configurations and sizes. Initial results from ongoing development demonstrate the binary compatibility with variable number of cores, as well as dynamic adaptation of schedules to data sets. We present preliminary results of key features for some benchmark applications.

Categories and Subject Descriptors

D.3.3 [Language Constructs and Features]: Concurrent programming structures; D.3.4 [Processors]: Run-time environments; C.1.3 [Other Architecture Styles]: Heterogeneous (hybrid) systems

General Terms

Languages, Management, Performance

1. INTRODUCTION

The advent of heterogeneous multicore architectures has had significant impact on the software infrastructure, notably programming models and software support. It has

been well established that specific HPC applications like N-Body Simulations, Molecular Dynamics, and Terrain Rendering [1] can experience order of magnitude or greater speedups when paired with architectures that are specifically tailored to their needs. Similar examples from the HPC community include the Los Alamos National Labs Roadrunner system. However, a consensus seems to have formed that the amount of effort required to port or rewrite applications to take advantage of these heterogeneous architectures without middleware support is non-trivial at best and herculean at worst. The need to tailor an application to a specific system architecture precludes cost-effective migration of an application from one system architecture to another making it difficult to amortize substantial investments in the development of code bases or to scale to larger system sizes. Consequently productivity is repeatedly traded for performance. This limitation motivates the need for runtime environments that allow applications to harmoniously utilize the heterogeneous resources offered by a system in a way that enables portability and scalability across a range of configurations.

Previous work on application development for heterogeneous many core systems has focused on case studies where the performance of particularly demanding applications is evaluated on highly specialized architectures such as FPGAs, GPUs, IBM Cells, and Intel IXPs, using customized solutions and vendor supplied tool chains. Programming language based efforts such as Brook [2] and StreamIt [3] encapsulate new stream programming models and focus on compiler-driven optimizations of these computing models on commodity platforms rather than heterogeneity. Notable exceptions include a new programming language for parallel systems with heterogeneous memory hierarchies [4] and efforts in workflow languages and coarse grain data parallel computing [5] that represent model driven approaches to leveraging existing languages and operating systems. Relatively few efforts have examined the more general problems of compatibility, scalability, and design complexity on heterogeneous platforms for generic applications, being content instead to address them for specific high profile applications.

In the context of these prior efforts, our approach is to address the performance-complexity gap by defining a programming model and associated execution model that is implemented by the Harmony runtime. The programming model is based on the identification of compute kernels, predicated kernel execution, and a managed shared address space. The execution model is based on dynamic detection and tracking of dependencies between compute kernels (enabled by the programming model), and a decoupling of

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.

HPDC'08, June 23–27, 2008, Boston, Massachusetts, USA.
Copyright 2008 ACM 978-1-59593-997-5/08/06 ...\$5.00.

kernel invocation and kernel scheduling/execution. The approach is inspired by solutions to instruction scheduling and management in out-of-order (OOO) superscalar processors, where those solutions are now adapted to schedule kernels on diverse cores. The result is binary compatibility across system configurations with at least one core in common and scalability maintained via a two step solution - flow dependencies are first inferred for a window of compute kernels that have yet to execute and then used as constraints to a scheduler that attempts to minimize the execution time of the application while satisfying all dependencies.

2. HARMONY PROGRAMMING MODEL

The core of the Harmony programming model is relatively simple: applications are composed of code segments that invoke *compute kernels* whose execution order may be subject to explicit *control decisions*. As kernels are encountered in the course of the execution the application, the kernels are dispatched to the runtime via non-blocking calls. Synchronization takes place when the application checks for the availability of the outputs of a dispatched kernel via blocking calls. All cores have access to a shared address space.

Compute Kernels are analogous to functions or procedure calls in imperative programming languages. However, this model places restrictions on compute kernels that enable fast inference of parallelism and independence from architecture. First, all input and output arguments and their memory locations must be known when the kernel is invoked, i.e., no pointers embedded in the input/output arguments. Second, a specific implementation of a kernel must use a single heterogeneous processor exclusively. Third, a kernel may have implementations for multiple processor architectures, but, if so, all implementations must be identical in the sense that they produce the same outputs for all possible combinations of inputs. Finally, kernels may allocate local memory for scratchpad computations, but this memory may not be persistent across kernel executions.

Control Decisions are syntactic structures for specifying control-dependent kernel execution and are analogous to branches in that they can potentially change the sequential flow of a Harmony application. Control decisions take in a set of input variables and determine the next compute kernel or control decision to be executed. Like compute kernels, control decisions can have multiple implementations for different architectures as long as they behave identically. The explicit specification of such control dependencies enables transparent optimizations such as speculation and predication to improve concurrency at the level of compute kernels in the same manner that branch prediction improves instruction level parallelism.

Shared Address Space enables Harmony to treat kernel arguments as global variables explicitly managed by the Harmony runtime. Intuitively, global variables can be thought of as regions of memory of arbitrary size that can be made visible to kernels, enabling runtime dependency checks. These checks preclude the need for hardware support for coherence while memory consistency requirements necessitate ensuring that all outstanding memory operations from a specific kernel have completed before the kernel signals to the runtime that it is finished.

2.1 Execution Model Example

The execution of an application implementing this model

may be viewed as logically equivalent to that of a sequential application executing on a modern super-scalar, out-of-order (OOO), processor with instructions replaced by kernels and functional units replaced by heterogeneous processors. During execution, an application (control) thread dispatches kernels via the Harmony API causing the kernel along with dependence information to be queued by the runtime for execution. Dependence-driven scheduling over kernels in a *dispatch window* deploys kernels on cores for which kernel binaries exist. Optimizations such as speculative kernel execution are supported via the separation of kernel execution (affecting local state) and commitment of the results of execution (affecting global state).

The following example is used to illustrate the advantages and potential of this model of kernel level parallelism, as well as identify opportunities and challenges. Figure 1 shows an example of a sample execution of two iterations through a H.264 video decoding application. Note that this example is intended to highlight the features of the complete execution model in a way that can be easily understood, and does not correspond to an application that we have actually implemented. For results from current applications implementing this model, please see Section 3.

Figure 1a shows the basic building blocks of the application; these building blocks correspond to compute kernels in the Harmony programming model. The complete application is shown in Figure 1b, where the large blocks represent kernels and control decisions, and the smaller blocks represent global variables (the darker variables represent inputs and the lighter variables represent outputs). In this example, the first kernel parses a frame from an input stream, updates the frame index, and stores the encoded frame in a temporary variable. The next four kernels perform each stage of the decode operation using temporary variables to pass the intermediate results. The final object in the program is a control decision that stores the decoded frame in an output video array and jumps back the beginning if the index has not reached the end of the input stream.

When the Harmony runtime begins executing this program, it scans through the kernels as they are encountered in the sequential flow of the application without blocking. As it is doing this, it builds a graph representing all of the data dependencies among kernels that have been encountered. Control decisions limit the number of kernels that can be scanned without blocking because they potentially change the flow of the application. This limitation can be skirted using techniques like prediction or predication to speculatively scan kernels beyond a control decision as long as there is a method for recovering from misspeculations. In Figure 1c, the white boxes denote kernels that have been invoked speculatively.

Another issue arise when two kernels share the same memory, but do not have a producer/consumer relationship. In Figure 1b, all of the temporary variables are reused through multiple iterations of the decode loop. However, the decode kernels do not have explicit flow dependencies. These are directly analogous to output and anti dependencies in instruction scheduling. Consequently, variable renaming is used to eliminate non-flow dependences: every time a non flow dependency is found between two kernels, the runtime allocates a new variable to be used instead, effectively breaking the dependency at the cost of increasing the memory footprint of the application. In the dependency graph shown in

Figure 1a. H.264 Decode

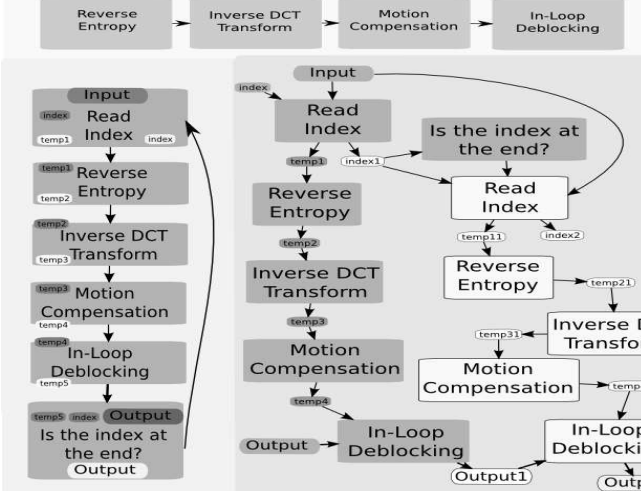


Figure 1b. Harmony Program

Figure 1c. Acyclic Dependency Graph

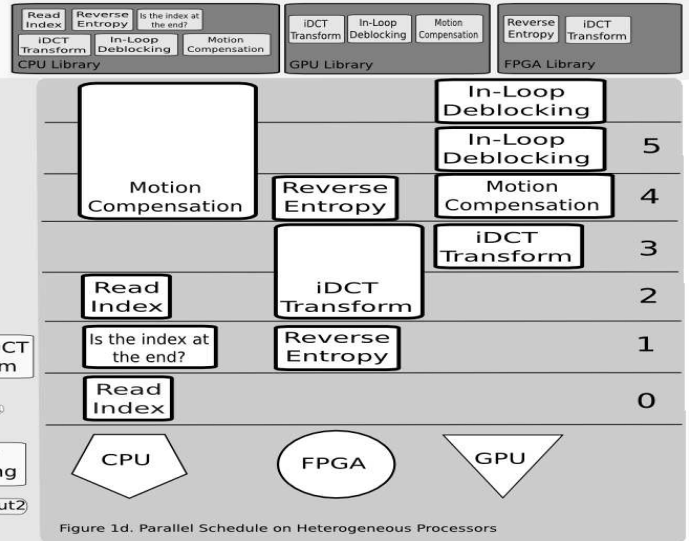


Figure 1d. Parallel Schedule on Heterogeneous Processors

Figure 1: Example execution of H.264 using Harmony

Figure 1c, all of the variables denoted in white boxes have been renamed and replicated.

The runtime utilizes a machine model description (e.g., number and type of cores) paired with libraries from the application to provide implementations for each kernel for at least one (but possibly many) ISAs. As kernels complete execution, dependencies between kernels in the dispatch window are updated and the schedule is revised. Figure 1d shows an example schedule for the H.264 application using a system with a CPU, GPU, and FPGA. This scheduling phase creates a dynamic mapping from kernels to heterogeneous architectures, while the existence of binaries for multiple machines permits execution on systems with radically different processor configurations, and performance to scale as more resources are added to a system until kernel level parallelism is exhausted.

Note that the execution time of individual kernels varies across architectures and can even be data dependent (in Figure 1d, motion estimation runs three times as long on the CPU as the GPU), yet being able to accurately predict this time a-priori improves the quality of the schedule. To address this issue, we plan to use online monitoring support for execution time of each kernel as well as values of some of the input variables. One can then construct data-value dependent models of execution time as a function of target core and utilize these to make more effective scheduling decisions. We also plan to investigate the use of programmer supplied management functions in a kernel to use domain specific information.

3. EXPERIMENTAL EVALUATION

Three sample applications were chosen to demonstrate the feasibility of applying the Harmony programming model across different domains: multiplication of dense matrices using a panel-dot algorithm, ray tracing, and audio frequency analysis. These applications were chosen to represent three classes of programs - (1) those with no data dependencies, (2) those with loop carried dependencies, and

(3) those with complex dependencies from domains that are relevant to scientific computing, computer graphics, and media processing respectively.

3.1 Experimental Setup

The experimental evaluation encompassed the following: (1) The test platform consisted of a AMD Athlon64 2.4Ghz CPU and an NVIDIA 8800GT GPU with 2GB DDR2 800 RAM, and Linux 2.6.22-14 as the operating system. All CPU kernels were written in ANSI C++ and compiled with GCC 4.1.3 and all GPU kernels were written using NVIDIA CUDA and compiled with nvcc V0.2.1221. (2) The runtime did not implement speculative execution. (3) The runtime did not implement analytical curve fitting, but did use complexity functions as described in Section 2.1.

3.2 Overview of Results

Figure 2a plots the normalized execution time of the matrix multiplication application using i) CPU only, ii) GPU only, and iii) CPU + GPU. Note that for small matrices, the CPU implementation is more efficient, but for large matrices, the GPU implementation is more efficient. This behavior can be explained with the observation that the GPU can do more floating point multiply operations per second than the CPU, but requires data to be sent to the GPU, processed, and the results to be sent back to the CPU. This operation can be amortized for large data sizes, but not small data sizes, resulting in the Figure 2a graph. The Harmony runtime is able to detect this behavior and transparently (to the application) move computation to the GPU as the matrix size increases, resulting in performance that follows that of the fastest single-processor application for all matrix sizes. This behavior is likely to be difficult to capture using a static partitioning without runtime support, particularly the crossover point where it becomes more valuable to use the GPU than the CPU, which is likely to depend on the exact model of CPU and GPU being used.

Figure 2b depicts the same normalized execution time for the ray tracing application, comparing the performance of

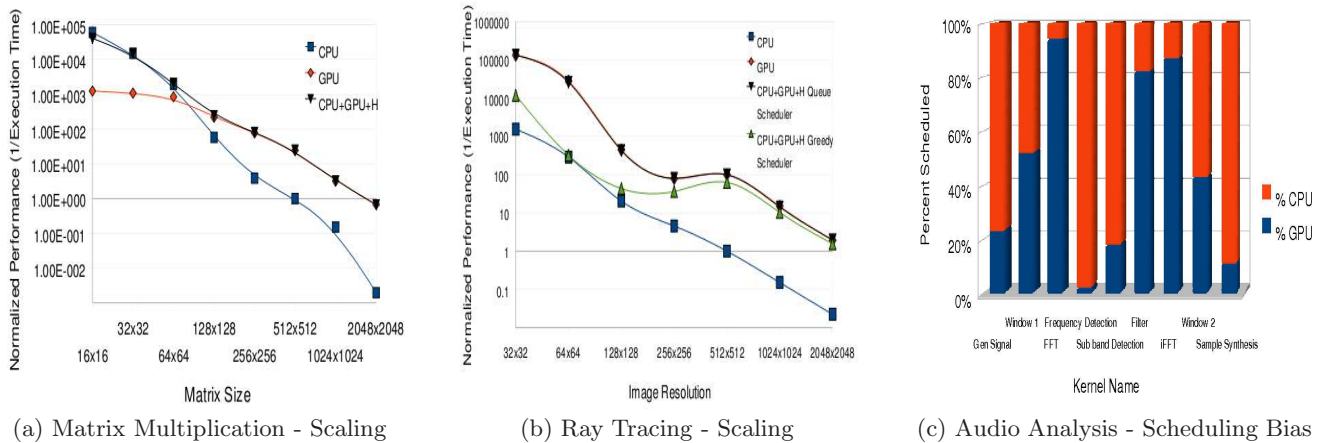


Figure 2: Application Results

the Harmony version with that of the standalone CPU and GPU versions. For this operation, the GPU is much more efficient than the CPU, experiencing a factor of 99.2 speedup. Such a dramatic difference in performance between the two versions of the kernel benefits greatly from a more intelligent scheduler. As the figure demonstrates, simply assigning kernels to cores as they become available as with a greedy scheduler performs as much as 80 times worse than a more sophisticated queue-based approach.

The final experiment demonstrates the difficulty in determining an efficient schedule for an application with many data dependencies. The audio frequency analysis application consists of nine different compute kernels that attempt to detect certain frequencies in an audio signal and remove others. Figure 2c shows the percent of time that each kernel was run (via automated Harmony decision making) on the GPU versus the CPU. In general, some kernels perform better on the GPU and others on the CPU, but there is no fundamental rule that can be used to make this determination statically. This result highlights the principle advantage of using a runtime like Harmony: an effective partitioning of work among cores can be unintuitive, data dependent, and system configuration dependent making it more amenable to dynamic runtime mapping than static allocation by an application developer or compilation environment.

4. CONCLUSIONS AND FUTURE WORK

The landscape of heterogeneous computing is dominated by two extremes: high performance and high software development complexity. In this paper, we have described ongoing work in developing Harmony: a runtime with the goal of reducing the complexity of application development while maintaining high performance that is scalable to many core systems with heterogeneous architectures. By using a compute kernel abstraction with constraints we find that we can leverage existing code bases, significantly improve portability and scalability, simplify development for sufficiently large classes of applications, and still harness much of the cost effective potential of heterogeneous architectures. In addition to the features described in Section 2, future work will explore: (1) variable granularity scheduling, where groups of ready kernels are automatically coalesced and scheduled as

a single entity, thereby trading scheduling overhead for concurrency, (2) hierarchical application, where the Harmony programming model is applied hierarchically to distributed systems of heterogeneous many core nodes, and (3) pattern recognition, where common kernel patterns are identified and their corresponding schedules are cached, forgoing the need to resolve dependencies and reschedule for repetitions of the patterns.

5. ACKNOWLEDGEMENTS

The authors gratefully acknowledge the generous support of this work by LogicBlox Inc. both through research grants as well as technical interactions, and equipment grants from Intel Corp. and NVIDIA Corp.

6. REFERENCES

- [1] V. T. Barry Minor, Gordon Fossum, “Terrain rendering engine (tre),” tech. rep., IBM, 2005.
- [2] J. Suh, E.-G. Kim, S. Crago, L. Srinivasan, and M. French, “A performance analysis of pim, stream processing, and tiled processing on memory-intensive signal processing kernels,” *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pp. 410–419, 9–11 June 2003.
- [3] M. Gordon, W. Thies, and S. Amarasinghe, “Exploiting coarse-grained task, data, and pipeline parallelism in stream programs,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA), Oct. 2006.
- [4] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, “Sequoia: programming the memory hierarchy,” in *SC ’06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 83, ACM, 2006.
- [5] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, “Merge: a programming model for heterogeneous multi-core systems,” in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 287–296, ACM, 2008.