A Multi-Paradigm Programming Model for Heterogeneous Architectures

Michael Champigny mchampig@mc.com
Mercury Computer Systems, Advanced Computing Solutions

Introduction

Microprocessor trends indicate a shift towards wider data paths, flattening clock frequencies, and increasingly complex memory hierarchies. Asymmetric architectures with a mix of complex out-of-order cores and hardware managed caches, fixed-function logic for data-parallel computations, and a larger numbers of simpler in-order cores with software managed caches appear to achieve an optimal balance of die area, power-performance and performance-cost ratios.

Mapping algorithms and application workflow to embedded and heterogeneous architectures poses several challenges to application developers. Data locality is a first-order concern in the presence of specialized accelerators such as graphics processors which must share the interconnect bus with chip multiprocessors. Explicit multi-threading and complex synchronization protocols employed to reduce the latency of compute-bound workloads are prone to oversubscription, race conditions, and deadlock. Existing industry standards, such as OpenMP for coherent memory architectures and OpenCL for heterogeneous architectures work well for many applications but have some shortcomings in the embedded domain. OpenMP requires compiler support which may limit broad applicability to some embedded targets, and favors workloads with highly structured memory access patterns such as parallel loop constructs. OpenCL has optional support for exposing parallelism to chip multiprocessors but has limited support for task parallelism, mostly due to the uniform task interface it must support on GPU devices which currently lack support for nested task parallelism and point-to-point synchronization.

We describe preliminary results of our research in parallel programming environments for embedded, heterogeneous platforms. Our solution for mapping algorithm kernels and applications to a mixture of CMP (chip multiprocessors), MCA (many-core arrays), and GPU (graphics processing units) accelerators is a software stack consisting of portable, ISO C++ language annotations for supporting multiple parallel programming model interfaces, a dynamic runtime scheduler for load balancing, and a hardware abstraction layer and component model for integrating specialized accelerators such as GPU devices. Our design choices were motivated by our goals to minimize latency, reduce jitter across multiple algorithm executions, and maintain acceptable scalability with evolving silicon roadmaps while improving productivity by supporting multiple parallel programming paradigms in a unified framework.

We demonstrate empirical performance results of our parallel runtime measured against the current state-of-the-art solutions in industry, namely OpenMP and Cilk Arts Cilk++. We chose these solutions for a comparative study due to their maturity in the industry, rich source of example algorithms, number of empirical studies, as well as their different algorithmic approaches. Our tests consisted of several numerical kernels operating on relatively small data sets common in embedded defense and digital signal processing applications. The tests reveal that latency, deterministic performance, and parallel efficiency were as good as, and in many cases exceeded, existing solutions with a similar measure of changes at the source code level. These results were achieved using COTS compilers and tooling and required no changes in the development workflow. In addition, there is no requirement to use proprietary or customized tools.

Architecture

As shown in Figure 1, our programming environment is a strictly layered software stack, allowing developers the flexibility to choose an appropriate level of abstraction based on their requirements. We avoid performance penalties at all levels of the stack by employing judicious use of template meta-programming, code in-lining, and the use of the pre-processor. At the highest level, application developers can employ the programming model interface that most closely matches the structure of a specific algorithm or workflow. A compiler back-end could target the runtime layer if a novel programming model is required. These choices can be made at any level of granularity throughout an application, with mixed paradigms common in most applications.

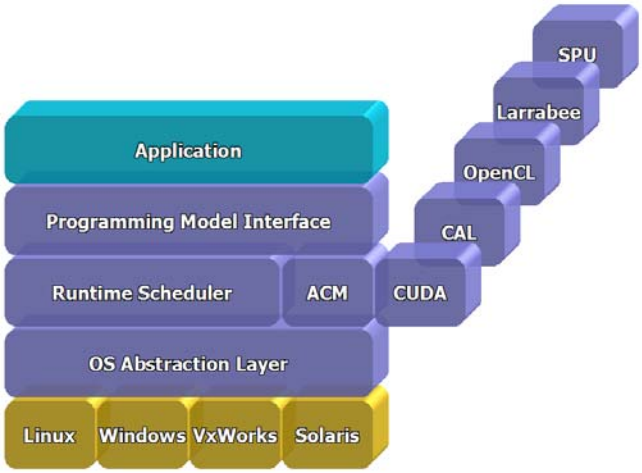

Figure 1: Parallel Programming Stack.

At the top of the stack is the programming model interface, currently consisting of a model for representing divide-and-conquer, nested task parallelism and a model for representing data parallel computation. These models present a different interface to a unified runtime layer

which the developer links their application to. A programmer annotates ISO C++ source code with macros operating as language extensions. These annotations expose fine-grained parallelism to the unified runtime layer. It is relatively straightforward to incorporate additional programming model interfaces in an additive way without disrupting the lower-level runtime. Porting existing kernels or workflows is mainly an exercise in choosing the appropriate programming model and mapping the source language annotations to the selected programming model interface. Sequential code may evolve incrementally by the insertion of the appropriate parallel language annotations. Any application may freely make use of more than one programming model interface. For example, at higher abstraction levels of an application, functional decomposition can expose coarse-grained task parallelism. At the lower levels, such as a loop block, it may be appropriate to switch to data decomposition and process subsets of the data in parallel.

The runtime layer consists of a dynamic, load-balancing scheduler. The scheduler employs a variation of a work-stealing algorithm and cache-reuse heuristics, proven elsewhere [1] to achieve optimal efficiency. The runtime is *work conserving* and executes fine-grained tasks without pre-emption, similar to proposed hardware approaches [2]. In close cooperation with the runtime scheduler, the ACM (accelerator component model) is used to integrate and manage throughput-oriented devices, such as NVIDIA and ATI GPU devices, Intel's Larrabee, as well as the Cell SPU complex. This component load balances compute kernels across multiple GPUs, even in multi-vendor configurations. As with any component model, additional components may be added to the system so long as they implement the expected interface for data transfer (such as DMA operations), computation offload, and module and kernel management.

Finally, an operating system abstraction layer is used to isolate the upper levels of the stack from platform-dependent features. Ideally, porting the entire stack to a new platform involves porting only this abstraction layer. Operating system features that are abstracted include thread management, synchronization primitives, memory page allocation, and performance-related hardware features.

Programming Model

Developers are exposed to the programming environment through parallel programming model interfaces augmented with future variables. These effectively isolate the design paradigm from the runtime implementation. We proved the efficacy of this approach by implementing several numerical algorithms with multiple programming model interfaces. Numerical algorithms better expressed in a data-parallel style proved more efficient and often took less lines of code than the divide-and-conquer representations, even though the underlying runtime layer is shared. Because our programming environment is compiler-agnostic, we leverage native vendor compilers optimized for use on specific processors for better performance. This proved advantageous for several numerical kernels and provided interesting comparisons between compiler implementations on the same platform. We plan on extending the number of programming model interfaces we expose in the future by lifting current restrictions on control flow in the computation graph.

Performance

For smaller data sets our runtime was more effective in terms of parallel efficiency and latency for compute-bound workloads. For larger data sets, results varied more widely across different runtimes and kernels. In all cases, the standard deviation across batched kernel executions was smaller in our runtime in comparison with OpenMP, Intel Threading Building Blocks, and Cilk++ which is likely the result of our focused optimizations for embedded, real-time algorithms. In Figure 2 we note the results of a dense square matrix multiplication with single-precision floating point dimensions of 256x256. On a dual-core CMP, a typical configuration for an embedded platform, we see that our scheduler outperforms all others at each tested tile size. This test illustrates the common trend we observed across many kernels, data set sizes, and compilers.

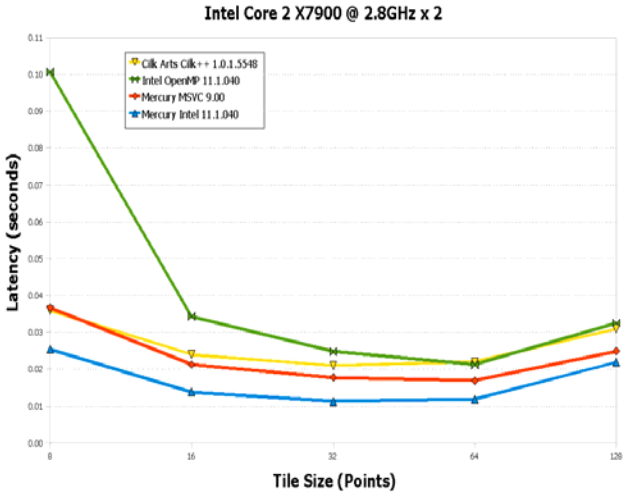

Figure 2: Dense Square Matrix Multiply Latency (64K-point).

Summary

We presented initial research findings and preliminary performance results of a parallel programming environment for embedded and heterogeneous platforms. Our approach leverages open or proprietary ISO C++ compilers to expose latent fine-grained task and data parallelism in applications and algorithms and map it to a variety of parallel architectures, such as chip multiprocessors, symmetric multiprocessors, and GPU accelerators. A distinguishing feature of our architecture is the inclusion of programming model interfaces to a single, unified runtime to improve productivity and to expedite the incorporation of existing parallel algorithms into our framework.

References

[1] U. A. Acar, G. E. Blelloch and R. D. Blumofe, *The Data Locality of Work Stealing*, Theory of Computing Systems, pg. 1-12, 2000.

[2] S. Kumar, C. J. Hughes, and A. Nguyen, *Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors*, ACM SIGARCH, Vol. 35, Issue 2, pg. 162-173, 2007.