Leveraging Semantics Attached to Function Calls to Isolate Applications from Hardware

Sean Halle
INRIA Saclay and UC Santa Cruz
Email: sean.halle@inria.fr

Albert Cohen
INRIA Saclay
Email: albert.cohen@inria.fr

Abstract

To improve performance, computer systems are forcing more microarchitectural and parallel hardware details to be directly exploited by application programmers, exposing limitations in existing compiler and OS infrastructure, which is failing to maintain the software productivity of the past. In this paper we propose a pragmatic approach, motivated by our experience with BLIS [11], for building applications that tolerate changing hardware, delivering good performance from the same source across diverse parallel targets. Applications are coded in terms of generic parallel patterns using a "piggy back" language, embedded into a base sequential lanuage by attaching semantics to function calls. Our approach allows programmers to leverage multiple processor-specific and domain-specific toolchains encapsulated in specialization modules, which extract their input information from the semantics of the function calls, creating an isolation layer bewteen application and target platforms. Developers use existing sequential development tools and languages to code and debug, performing specialization as a separate step when shipping the code. We show this approach can successfully specialize a single source to diverse and evolving heterogeneous multi-core targets and enable aggressive compiler optimizations.

1 Motivation and Overview

Programmers and project managers ask "what parallel language should I port my application to?" They want to port it once, and not port it ever again, only touching the code afterwards for application-driven reasons, but not for hardware or performance reasons. This can only be achieved with an isolation layer that takes away control of deciding when and where a task runs, hiding these decisions in the *isolation layer*. To be efficient on hardware, the isolation layer also needs control over the work in a task, by fusing or fissioning task code, and/or by controlling the size of data or iteration space in the task. Finally, to enable the lowest overhead scheduling, the isolation layer needs to be supplied with a model that predicts execution time and variability of a task, to help it make task-size and placement decisions.

Most programmers want to know how their code choices and factoring of the problem will affect performance, even when their code will be transformed by tools in ways they cannot even imagine, to run on hardware they can't even imagine at development time. In addition, project managers want the code to be well organized for future code-maintenance and feature enhancement work, and easy to generate the executables for future hardware targets. This requires having a methodology, with supporting tools organized into a platform, to go with the isolation layer.

In this paper we propose an isolation layer and methodology, with platform, motivated by our experience with BLIS [11]. It uses a "piggy back" language embedded into a base language by attaching semantics to standardized function calls, to express generic patterns in which application code interacts with scheduler manifestations. One pattern is data-structure centric similar in spirit to SPMD programming called *DKU*, another, called *WorkTable*, covers arbitrary communication, control, and transactions, in the spirit of software-components with added parallel semantics.

These two allow essentially any program to be composed hierarchically, capturing the information that parallelizing and run-time scheduling tools need, such as dependencies, boundaries of fusable function-blocks, constraints on accessing shared data, and constraints on overlap of tasks. The application supplies "plug-ins" that *communicate directly with the scheduler*, to implement constraints and do things like change task data-size or task iteration-space size during a run.

The isolation layer is composed of modules, one for each target hardware, that contain tools applied to the source in a *separate specialization step* just before distributing the product. Development takes place in a sequential environment, where compile and debug are done until the application is complete. Then the specialization step applies the modules for desired targets, generating an executable for each, using source-to-source transforms and compilers embedded in the modules. This separation gives the productivity advantages of SeJits [5], but with the greater performance available with static analyses and transforms.

In essence, *the old mental model of programming to hardware is replaced by the new mental model of programming to the standard tool needs*. When developing, the programmers can compare coding choices for performance impact by knowing what information the tools need. For example, the finer the grain of dependency and task they can specify, the more parallelism opportunities are available to the tools, likewise, the more places they can enable overlap of tasks, the more parallelism choices the tools have.

The piggy-back language uses function-calls for syntax, that look like library calls, except that multiple functions are grouped into a pattern according to a grammar, with a grammar-checker and execution model included, making it a bona fide language. To provide the manipulators, some of the function-calls go in the reverse direction, being implemented by the application and called (back) by the platform.

The platform provides a base implementation of the piggy-back language for each hardware target, with a run-time system that does dynamic scheduling. During a run, the scheduler tunes task size by calling the application-provided manipulators. It also has available "overrides" generated by static tools, that have their own scheduling code that takes control of the hardware for a specific portion of the application. Domain-specific frameworks and hardware-specific tools filter the code to recognize portions they can do very high performance transforms on, then generate the override and a performance model the scheduler uses to decide which version is best under the dynamic circumstances.

For practical performance reasons, application developers may also write hand-tuned, non-portable, implementations of selected hot-spots in an application for specific hardware. They develop with 3rd party hardware-specific tools then wrap their code as an override and provide a cost model for the platform's dynamic scheduler to use.

Section 2 follows an example application. Section 3 presents observations from hardware, compilers, and parallel programming practices that motivate the proposed approach, while Section 4 discusses practical aspects and Section 5 displays evidence of the approach's potential. Section 6 summarizes the lessons learned.

2 Methodology

H264 video decoding is used as a running example. Here a frame is (essentially) composed of macro blocks, which undergo computation of their motion vectors, followed by gathering the target pixel information and combining it with the the macro-block's pixel information. The result goes to a deblocking filter, after which it *may be* the target of motion vectors in succeeding frames. Motion vector calcuation depends on macro-blocks in a 30 degree diagonal above and left, as well as possibly several previous frames. Meanwhile Deblocking depends only on the macro blocks, in the same frame, above and left.

This can be encoded using BLIS's WorkTable patterns, which encode a graph where work-units flow. The nodes are either application code, called *work-piles*, or one of the forms of scheduler node, which take plug-ins provided by the application. The plug-ins perform the application-to-scheduler communication.

In H264 a *WorkDivider* scheduler node would be placed at the entrance to the graph, and given a plug-in that divides each video frame into individual macro-blocks, to which it adds an ID and *bookkeeping data*. The 30 degree motion-vector dependencies are enforced by placing a *HoldUntil* scheduler node just before the work-pile that calculates the motion-vectors. It holds macro blocks back until all propendent macro blocks have appeared at the exit of the motion-vector work-pile. The app supplies the HoldUntil a plug-in that specifies the location to watch and calculates the IDs of propendents to watch for.

The completed motion vectors fetch their target macro blocks, stored in a *Keeper* which keeps around data that *might* be used at some point, much the way databases do. This fetched pixel data gets paired up with the inverse transformed pixel data using a *ReJoiner* to make sure the motion vector data and inverse transform data are for the same macro block.

Such an approach covers much more than just embarassingly parallel algorithms; in contrast to MapReduce [6], it allows a single arbitrary data structure, such as a mesh, to be divided into pieces that communicate with each other, and so cleanly handles the same kind of irregular problems as Galois [14].

The ReJoiner and, at the exit of the graph, the *WorkUndivider* accomplish the same logical effect as a join or barrier operation, but without causing idle time. Further, partial results are available, to be used in parallel with the still-in-progress computation.

The simpler, DKU, pattern may also be used. In H264 it has been applied to deblocking [1], where each 45 degree diagonal is made available for sub-division.

The graph makes available, to tools, the flow of data, and the HoldUntil and ReJoiner functions supply overlap-of-computation and access-of-shared-data information. This is useful for efficient hardware usage, for Formal Verification of race conditions, and for exposing difficult to recognize parallelism to tools.

The WorkTable (WT) patterns have the software engineering benefits of software components. Standard sequential programming takes place within the function for a work-pile. Only at the point when composing piles in a graph are inter-function and inter-data dependencies considered, adding HoldUntil and ReJoiner piles to establish a partial-ordering.

Returning to the H264 example, in the reference code the dependencies are enforced by completing entire frames in-order. With BLIS's WorkTable, the order of completion is data-driven. As soon as a macro block finishes deblocking, it becomes available immediately to both motion compensation and the following diagonal, exposing higher parallelism [3]. To ensure correctness, the ID attached to each work-unit, plus the plug-ins, get used to state the partial ordering in a natural way that makes intuitive sense.

This degree of parallelism, due to overlap of partial results from multiple loop nests, is unavailable to approaches that place scheduling inside the tools but don't provide direct interaction between scheduler and application code, including OpenMP [17], OpenCL [9], Sequoia [8], Cilk [10], and Map Reduce. It can, of course, be exploited using Threads (like TBB [7]) or message passing (MPI [16]), or Merge [15], but at the cost of losing portability, due to exposing details such as pinning Threads to cores, location of results, current load on cores, size of productive task, structure of Kernel and so forth, which change with target.

CnC [13] can express this level of concurrency for some applications, but not H.264 due to the motion-vectors, which are data-determined dependencies. In addition, debug is difficult as CnC's translation tool is in the compile-test-bug-fix cycle, plus CnC requires more extensive modifications of the base serial code to convert loop-nests to tag-collections and eliminate side-effects, which BLIS allows, and some nests in H.264 do not fit CnC's straight-forward dependency structures.

The proposed platform is designed to support the static tools in the specialization modules. For example, the WT patterns make work-pile function boundaries and communication among them available for task-enlarging decisions. As another example, the data-flow graph can be extracted and transformed to the input format of a domain-specific and HW specific tool like that of Sundaram et al. [19] that finds a near-optimal scheduling of GPU kernels for large data-sets. Kernel optimization tools [4][18][2] can then be placed into the same specialization module, to improve the individual kernels. Such tools may also be tuned at the time they get added to a module, by running experiments to gather parameters such as copy-to-GPU time, network bandwidth, or performance of data-layouts in auto-tuners. Because the specializaiton step is outside the development cycle, longer-running builds that use profile-driven techniques and iterative optimization become practical.

For decisions better made at run-time, the same information supports the dynamic scheduler, providing it with the communication pattern and performance models, to predict locality and better overlap communication. When not enough tasks are available, the app-to-scheduler interaction approach allows in-progress tasks to be taken back and re-divided, as demonstrated in DKU on Hamiltonian Path [12], significantly improving load-balance, especially on NP and other difficult-to-predict problems.

As another example of application-to-scheduler interaction, real-time applications may ask the isolation layer for decisions that involve hardware configuration and availability. For instance, it can ask what size of data will take a specified amount of time to complete, in a kernel with supplied performance predictor. This allows latency-sensitive applications to build a WorkTable graph suited to the machine, without knowing HW details. In this example, it would build a tree of work-piles for ever larger data-blocks, adjusting fan-out to HW. Such scheduler-to-app interaction provides a HW neutral interface that can handle latency requirements in complex ways.

3 Fundamentals of the Problem

At the heart of performance portable parallelism lies the modification of the patterns observed in a running application, to fit them onto the structures in the hardware. As a simple example from sequential tools, loop unrolling *modifies the code's use of registers* to express multiple of the same operation, to increase utilization of wide-issue function units. Likewise, loop interchange *modifies the sequence of visitation of points in iteration space* such that addresses cluster differently, reducing the number that map to the same cache location between re-uses, to reduce conflict misses.

The same holds true in the parallel world: performance improvements come from re-arranging the patterns of operations, and addresses, for a "better" fit to hardware structures. The main difference between sequential and parallel optimization lies in the size, and so scope, of the state-evolution pattern that must be considered, and in the scheduling process.

Parallel hardware forces larger-scale structures in the application-evolution pattern to be considered, due to the larger scale of parallel hardware structures, and the larger overhead of scheduling onto them. In a nutshell, the larger the communication time plus scheduling overhead, the larger the scale of application-evolution pattern required to realize speedup on the hardware structure.

Hence, the piggy-back language includes interactions between scheduler and application which *enable modifications of the application's state-evolution patterns* in ways otherwise unavailable to the run-time. Due to the interaction, the dynamic scheduler gains the ability to choose state-evolution patterns that fit best to the hardware size and state.

For example, in the DKU pattern, the Divider subdivides a DKU "task" into smaller ones. The isolation layer calls the Divider, telling it how many sub-tasks to make. Because the work performed by a Kernel, or the function attached to a work-pile, is often dominated by a loop-nest, the work in a task (*work-unit*) is often encoded as start and end values of iteration variables. Hence dividing the work just means divying up the start and end values. The effect on the evolution of state can be visualized by the addresses generated by a work-unit, where the size of the stream of addresses is set by the start and end values.

On a multi-core machine with two levels of cache, the dynamic scheduler calls the divider twice on large data structures. The first time it makes streams containing the same number of unique data-addresses as fills up the L2 cache with data. The second time subdivides each of those streams into smaller ones that fill only the L1 cache of each core.

This demonstrates how dynamic schedulers, which implement a portion of the piggy-back language's semantics, use the plugins implemented by an application to adjust the size of patterns in the state-evolution to fit well to hardware structures.

The scheduling process lies at the heart of parallelism. A close look at the hand-tuning of parallel code reveals that most tuning work involves modifying the scheduling of work-units (tasks), where a work-unit consists of a snippet of code plus data to apply the code to. The hand-tuning often changes the code in a work-unit, such as when going from multi-core code to GPU code. Multi-core hardware favors large complex kernels that perform as much computation as possible between inter-thread synchronization events. In contrast, GPU hardware favors small, highly regular kernels that run well in SIMD fashion. Thus, hand-tuning breaks up the single multi-core kernel into multiple GPU kernels. It also changes the scheduling process by changing pthreads synchronized with locks into GPU kernel invocations.

This motivates our choice to place scheduling inside the isolation layer and express in application code only the *constraints* on scheduling. The WorkTable patterns specify constraints via plug-ins, to scheduler nodes, that calculate the IDs of propendent work-units, and the DKU pattern implies them inside the DKUPieceMaker and uses an explicit call to the scheduling process to intiate the work, keeping the actual scheduling under the isolation layer.

The patterns trade the programmer need for natural expression of an application against the tool needs for deduceable scheduling constraints, resulting in neither the most natural expression of the application nor the most straightforward expression of scheduling constraints, but effective for both sides of the isolation layer none the less.

4 Practical Considerations

BLIS handles the practical aspects of development, including quality of tools, the experience of debugging, and learning curve, by using existing sequential development tools with existing languages. The development cycle of code-compile-test-bug-fix remains the same as current practices, including *symbolic debugging of the original source*. BLIS makes just three changes: 1) a BLIS directory is added to the application source tree and becomes part of the build 2) debug is split into three phases to isolate types of bugs from each other, and 3) an additional specialization step gets performed once development is complete, just before shipping.

The BLIS directory supplies the behavior needed to debug the parallel-patterns, within the sequential environment, by supplying an implementation of the piggy-back language that divides debug into phases for, functionality, then distributed memory, then race conditions. The directory also isolates hardware details that may differ between development vs target machine, such as pointer size and data alignment.

BLIS adds a separate specialization step in which a completed application gets sent to a server that applies a specialization module for each target hardware configuration, each of which generates its own executable. Install-clients on target machines extract the appropriate executable, hiding the multiplicity behind the familiar "install shield" method used today. Software-development firms may maintain their own server and provide distribution, or leverage a centralized one, presumably provided by a not-for-profit organization that charges a nominal fee for the service and enforces a workable standard for wide compatibility, which has been a challenge for past standards.

The specialization modules have been designed for easy integration of stand-alone tools, both commercial efficiency tools and current research, which would communicate with the other tools in the module via pipes or files. We hope to make an environment that simplifies performing research on specialization and scheduling by freeing research teams from the onerous task of building their own infrastructure. The growing suite of applications written for the platform by end-users would provide a realistic suite with which to measure the effectiveness of proposed approaches. We hope to found a non-profit that would provide the final polish on such tools, making them industry-ready and releasing them open-source, with authors' permission, as part of the platform.

We recognize that the BLIS approach of attaching semantics to standardized function calls, especially for application-to-scheduler interaction, represents a compromise between automatic tuning and manual hand-tuning. The application programmer expends extra effort to implement the plug-ins that perform scheduler interaction, while in compensation they receive, automatically, decent performance on all viable target hardware. As more tools become integrated into the platform, the performance of the same application code will increase, and performance of BLIS library functions will increase. Meanwhile the need to override hot-spots with hand-tuned code for specific hardware will decrease.

5 Demonstration of Effectiveness

Figures 1 and 2 show efficiency of hardware-usage across machines with differing numbers of threads. To allow direct comparison, the graphs use "percent ideal" speedup, defined as $\frac{\frac{T_S}{T_P}-1}{p-1} \cdot 100$, and plot the percent of ideal against sequential execution time over ever-larger problems. One of these graphs gives the relevant numbers: time of sequential execution, time of parallel execution, time of execution at break-even, size of input, size of input at break-even, absolute speedup, percent of linear speedup.

Time of sequential execution is read directly off the x-axis; time of parallel execution time is calculated, by converting percent linear into absolute speedup then dividing the sequential time by that; time at break-even is the zero crossing (this definition makes slowdown a "negative speedup"); and size of input data is found by matching data-points along the curve to the sizes stated in the caption.

In the experiments, we used the following machines: a one-socket 2 core, denoted "1x2", a 2 socket by 4 core each, denoted "2x4", a 4 socket by 4 core each, denoted "4x4", and a heterogeneous collection of those connected by 100Mbit LAN, which demonstrates the use of both hierarchical division and distributed memory, from the same source.

Figures 1 and 2 show efficiency of HW usage across the four machines on the same two sources written in Java with BLIS's DKU pattern used.

For Matrix Multiply, break even matrix size for the multicore machines lies at around 100x100 cells, double precision, and at around 200x200 on the LAN-connected collection. The small-matrix slow-down will be avoided when an execution-time predictor is supplied, allowing the schedulers embedded by the specialization modules to predict performance loss and choose the serial version.

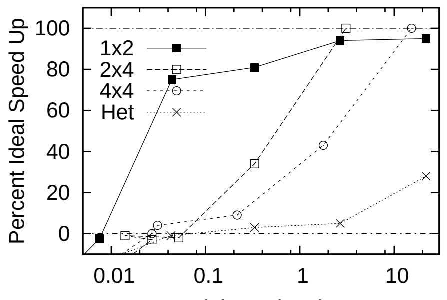

Figure 1. Efficiency of HW usage across four machines for Matrix Multiply, on matrices of size: 9x9 (not seen), 81x81, 162x162, 324x324, 648x648, 1296x1296

All input graphs for Hamiltonian Path have no path, so the amount of work stays constant across machines [12]. The plot shows that the speedup approaches perfect on each machine once the serial time becomes large enough.

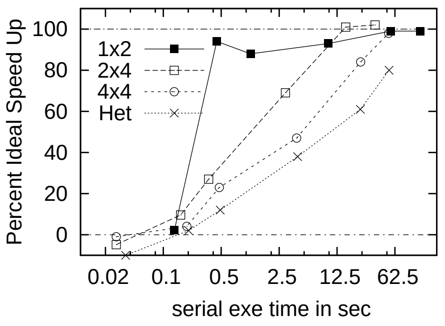

Figure 2. Efficiency of HW usage across four machines for Hamiltonian Path on input graphs of size: 14, 16, 18, 20, 20, 20, all have no path.

These figures indicate that applying specialization modules, containing discrete command-line tools that automatically generate multiple executables from a single source written with generic patterns, successfully delivers decent performance.

6 Conclusion

A hardware-isolating platform should, 1) have a specialization phase that analyzes and transforms the code, and uses or inserts a dynamic scheduler that can further transform the state-evolution patterns to fit onto hardware patterns; 2) end the thread programming model in applications, by hiding it below the isolation layer, which exposes only scheduling *constraints*; 3) state scheduling constraints, by implied semantics attached to function-call names (or keywords) and by explicitly stating code or rules for overlap of work; 4) make state-evolution transform plug-ins available to specialization; and 5) have application code explicitly interact with the isolation layer and its scheduling process, by making them first-class entities.

For practical reasons, such a platform should 1) reuse existing sequential environments and languages, for highly productive development and debugging; 2) provide generic all-inclusive patterns that deliver decent performance on the whole application on all hardware, and also provide a gentle learning curve, for incremental performance improvement; 3) absorb third-party tools, even ones for domain-specific patterns and hardware-specific transforms that only boost performance on some portions of an application for some hardware; and 4) automate the specialization step, for example by collecting the tool chain for each hardware configuration into a standalone specialization module, and using an install-client that runs on the end-hardware to select the correct executable, hiding the existence of multiple versions.

Bibliography

[1] DKU-ized Deblocking Filter code. http://dku.svn.sourceforge.net/viewvc/dku/branches/DKU_C__Deblocking__orig/.

[2] Spiral Group at CMU. Spiral homepage. http://www.spiral.net.

[3] Arnaldo Azevedo, Cor Meenderinck, Ben Juurlink, Andrei Terechko, Jan Hoogerbrugge, Mauricio Alvarez, and Alex Ramirez. Parallel h.264 decoding on an embedded multicore processor. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 404–418, 2009.

[4] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, 2008.

[5] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. *First Workshop on Programmable Models for Emerging Architecture at the 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[6] Google Corp. MapReduce home page. http://labs.google.com/papers/mapreduce.html.

[7] Intel Corp. TBB home page. http://www.threadingbuildingblocks.org.

[8] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, 2006.

[9] Kronos Group. OpenCL home page. http://www.khronos.org/opencl.

[10] Cilk group at MIT. CILK homepage. http://supertech.csail.mit.edu/cilk/.

[11] Sean Halle and Albert Cohen. BLIS website, November 2008. http://blisframework.org.

[12] Sean Halle, Dmitry Nadezhkin, and Albert Cohen. A framework to support research on portable high performance parallelism. http://www.soe.ucsc.edu/share/technical-reports/2010/ucsc-soe-10-02.pdf.

[13] Kathleen Knobe. Ease of use with concurrent collections (CnC). In *HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism*, March 2009.

[14] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, 2007.

[15] Michael D. Linderman, James Balfour, Teresa H. Meng, and William J. Dally. Embracing heterogeneity parallel programming for changing hardware. In *HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism*, March 2009.

[16] open-mpi organization. Open MPI home page. http://www.open-mpi.org.

[17] OpenMP organization. OpenMP home page. http://www.openmp.org.

[18] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 90–100, 2008.

[19] Narayanan Sundaram, Anand Raghunathan, and Srimat T. Chakradhar. A framework for efficient and scalable execution of domain-specific templates on gpus. *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, 2009.