Final Report

Award number: DE-FG02-11ER26044/DE-SC0006875

Recipient: The College of William and Mary

Final Project Report: Data Locality Enhancement of Dynamic Simulations for Exascale Computing

PI: Xipeng Shen (xshen5@csc.ncsu.edu)

Date: 4/27/2016

Period: 08/15/2011 - 08/16/2014

1 Review of Project Background and Goal

The development of modern processors exhibits two trends that complicate the optimizations of modern software. The first is the increasing sensitivity of processors' throughput to irregularities in computation. With more processors produced through a massive integration of simple cores, future systems will increasingly favor regular data-level parallel computations, but deviate from the needs of applications with complex patterns. Some evidences are already shown on Graphic Processing Units (GPU): Irregular data accesses (e.g., indirect references A[D[i]]) and conditional branches are limiting many GPU applications' performance at a level an order of magnitude lower than the peak of GPU.

The second hardware trend is the growing gap between memory bandwidth and the aggregate speed—that is, the sum of all cores' computing power—of a Chip Multiprocessor (CMP). Despite the capped growth of the peak CPU speed, the aggregate speed of a CMP keeps increasing as more cores get into a single chip. It is expected that by 2018, node concurrency in an exascale system will increase by hundreds of times, whereas, memory bandwidth will expand by only 10 to 20 times. Consequently, data movement and storage is expected to consume more than 70% of the total system power. Bridging this gap is difficult; the complexities of modern CMP memory hierarchy make it even harder: Data cache becomes shared among computing units, and the sharing is often non-uniform—whether two computing units share a cache depends on their proximity and the level of the cache. On the recent IBM Power7 architecture, for instance, four hardware contexts (or SMT threads) in a core share the entire memory hierarchy, all cores in one chip share an on-chip L3 cache, and cores across chips share L3 and main memory through off-chip connections.

These two trends complicate the translation of computing power into performance, especially for a program with either intensive data accesses or complex patterns in data accesses or control flow paths. Unfortunately, both attributes present and will persist in a class of important applications. For instance, many scientific simulations deal with a large volume of data. And meanwhile, as most real-world processes are non-uniform and evolving (e.g., the evolution of a galaxy or the process of a drug injection), both the computations and data accesses of these programs tend to be irregular and dynamically changing.

Currently, the lack of support to these applications on modern CMP severely limits their performance. On GPU, as our recent study shows and other studies echo, performance enhancement of a factor of integers is possible when memory accesses or control flows are streamlined for a set of GPU applications. On multicore CPU, our studies show that traditional locality enhancement, for being oblivious to the new features of multicore memory hierarchy, may even cause large slowdown to data-intensive dynamic applications. The severity of the issues is expected to worsen as the two hardware trends continue.

Some recent studies try to match software with the trends, but in a limited scope or manner. For irregularities on GPU, most studies focus on irregularities analyzable through static analysis (e.g., data accesses in regular loops). Dynamic irregularities are harder to address because the needed analysis and transformations typically have to happen at run time. Some other research resorts to hardware extensions, an actual adoption of which is unclear for the entailed space cost and complexity.

For data locality, recent years have seen some exploitations of the new memory hierarchy on multicore for performance, but most of them are on process or thread scheduling, rather than program transforma-

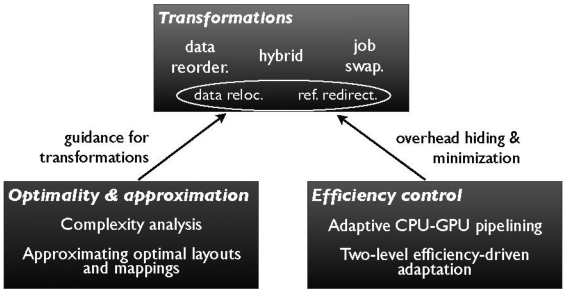
Transformations
data reorder.
hybrid
job swap.
data reloc.
ref. redirect.
guidance for transformations
overhead hiding & minimization
Optimality & approximation
Complexity analysis
Approximating optimal layouts and mappings
Efficiency control
Adaptive CPU-GPU pipelining
Two-level efficiency-driven adaptation

Figure 1: Major components of GStreamline.

tions. Our recent study reveals that program-level transformations may magnify the scheduling benefits by a factor of seven, concluding that program-level transformation should play a central role for data locality enhancement on modern CMP. But research in this direction has been sparse, and most have focused on data layout or cache performance modeling, rather than program transformations to match with the new memory hierarchy features. Overall, it is still an open question how to bridge the gap between dynamic computations and the two prominent properties of modern processors.

The goal of this project is to develop a set of techniques and software tools to enhance the matching between memory accesses in dynamic simulations and the prominent features of modern and future CMP, alleviating the memory performance issues for petascale and exascale computing.

2 Progress

The period covered by this report is the first three years of this project. We have achieved progress in the following aspects.

2.1 GStreamline for Runtime Memory Optimizations on GPU

First, we investigated the memory reference bottleneck on GPU computing. We developed a pipelined framework to remove irregular memory references on the fly. The framework is named *GStreamline*. It is a library for enabling runtime remapping between GPU threads and data elements so that irregular memory references can become regular. It consists of three components. As Figure 1 shows, its top component addresses issues on the creation of a new thread-data mapping. It contains a set of transformation methods built from two primary mechanisms, data relocation and reference redirection. The bottom two components address two fundamental questions for the thread-data remapping: the computation of desirable data layouts and mappings, and the minimization and concealment of runtime transformation overhead. Compared to the prototype we had before, GStreamline has a more sophisticated pipeline scheme, supporting fine-grained overlap between transformation and computation and between transformation and data transfer.

We evaluate the library on an NVIDIA Tesla 1060 hosted in a quad-core Intel Xeon E5540 machine. We experiment on seven programs that exhibit complex dynamic memory access patterns. These programs come from real applications, ranging from partial differential equation solvers (*3dlbm*) to conjugate gradient method (*cg*) to particle dynamics simulation (*hoomd*) to dynamic programming (*pathfinder*). GStreamline leads to 8–90% speedup to five of the programs; the heavy data dependences in *cfd* and *streamcluster* impose challenges to the optimizations, but the runtime adaptive control successfully prevents the optimizations from causing any slowdown. The results have been published at ASPLOS 2011 and ICS 2010 conferences.

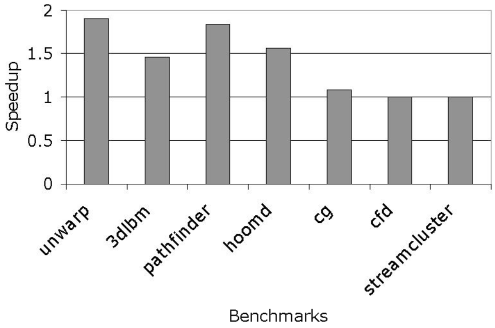

Figure 2: Speedup brought by GStreamline on NVIDIA Tesla 1060. The baseline is the performance of the original GPU programs.

2.2 Towards Optimal Data Layout and Placement for GPU Computing

The optimizations by *GStreamline* produce substantial performance benefits, but it has been relying on heuristic methods for computing appropriate data layouts. Some fundamental questions remain open: how to find optimal data layouts, how to approximate them with guarantees, and how to characterize the corresponding computational complexities. Answers to these questions will both deepen current understanding on the use of data layout adjustment for GPU optimization, and also shed insights for its deployment in practice. This is where the second fold of progress of this research happens.

An optimal data layout minimizes the number of memory transactions on the offchip global memory. We currently concentrate on memory coalescing on GPU; effects of on-chip cache are yet to be considered.

Through a reduction of the 3D Matching problem, we prove that if no extra space is allowed, finding an optimal data layout for a GPU kernel is NP-complete. We provide a 1-0 integer programming formalization of the optimal data layout problem. The formalization lays the base for designing approximation algorithms. We propose a series of approximation algorithms with different tradeoffs among space, layout optimality, and transformation overhead. All these algorithms have a certain guarantee on the quality of the resulting data layouts. The results have been published at MSPC 2012.

Later, we construct a complete system named PORPLE for optimizing GPU data placements. PORPLE consists of a mini specification language, a source-to-source compiler, and a runtime data placer. The language allows an easy description of a memory system; the compiler transforms a GPU program into a form amenable to runtime profiling and data placement; the placer, based on the memory description and data access patterns, identifies on the fly appropriate placement schemes for data and places them accordingly. PORPLE is distinctive in being adaptive to program inputs and architecture changes, being transparent to programmers (in most cases), and being extensible to new memory architectures. Our experiments on three types of GPU systems show that PORPLE is able to consistently find optimal or near-optimal placement despite the large differences among GPU architectures and program inputs, yielding up to 2.08X (1.59X on average) speedups on a set of regular and irregular GPU benchmarks. The results have been published at Micro 2014.

2.3 Asynchronous Data Transformations and Adaptive Control

In the third aspect, we explore the usage of co-processors, particularly GPU, for offloading expensive data layout transformations for dynamic simulations running on multicore CPUs.

Runtime data layout transformation has been commonly used for optimizing memory performance for dynamic simulations. However, the power of the transformations has been restrained by a dilemma. In all prior techniques, the runtime data or computation reordering happens synchronously—that is, the reordering is on the critical path of the application. This feature results in a tension between transformation quality and runtime overhead: More sophisticated transformations often yield better locality and save more execution time, but at the same time, they add more transformation overhead to the overall execution. The overhead can be substantial, especially for sophisticated transformations. For instance, one application of RCB—a classic data transformation approach—may take more than 20 simulation time steps. Moreover, the transformation has to be applied repetitively due to the iterative computations in dynamic simulations. Some studies propose to apply the transformation occasionally rather than every time when access pattern changes. Unfortunately, it is subject to the same quality-overhead dilemma: The less frequently the transformation applies, the less overhead it causes, but the worse the data layout is.

we propose three orthogonal techniques to resolve the quality-overhead dilemma.

The first is *asynchronous data transformation*, supported by a dependence-circumventing decomposition. The basic idea is to hide the transformation overhead by offloading the main transformations from the critical path, making them happen asynchronously (on an idle processor) in parallel with the execution of the application. Despite the simplicity of the idea, to the best of our knowledge, asynchronous data transformation has not been proposed previously. The plausible reason exists in the circular data dependences between data transformations and the execution of the application. On one hand, the transformation modifies the data structure that the application needs to read; on the other hand, the transformation needs to read some results computed by the application to figure out the appropriate data order. So, inherently, one invocation of a data transformation must run serially with the corresponding iteration of the application. In this work, we circumvent the problem by decomposing data transformation into two parts and safely relaxing some dependences through a careful analysis and layout approximation.

The second technique we develop aims at overhead minimization. It is useful when the system is equipped with massive parallel devices (e.g., GPU). We propose a novel data transformation algorithm, named *TLayout* (*T* for throughput), which reduces transformation overhead significantly with little compromise to the resulting quality. Unlike traditional data transformation algorithms, TLayout is a massively data-parallel algorithm, specially customized to the strengths of throughput-oriented co-processors. It is novel in using an almost dependence-free approach to grouping nodes into a number of clusters such that the nodes referenced adjacently fall into the same cluster. The algorithm shows high efficiency and good scalability.

Asynchronous data transformation and TLayout tackle the limitations of previous data transformations in two orthogonal directions; one for overhead hiding, the other for overhead minimization. Together, they help resolve the quality-overhead dilemma that prior approaches have been facing.

The third technique we develop is an online adaptive scheme. By transparently selecting the appropriate transformation strategy during runtime, the scheme gains the best of both asynchronous and synchronous transformations, proving able to overcome the limitations of both strategies.

Overall, the proposed techniques yield 65% higher performance improvement than previous techniques do, accelerating the original dynamic simulations by as much as a factor of 3.1 (2.4X on average) on five representative dynamic simulation benchmarks. Details of this work have been published at PACT 2011.

2.4 GPU-CPU Translation for Whole-System Synergy

In the final aspect, we have explored some correctness and efficiency issues in automatic translation from GPU code to multicore CPU code. The translation creates code portability across CPU and accelerators, critical for removing the need for device-specific code rewriting and promoting cooperations among different devices.

In the first part of this work, we concentrate on some correctness issues in compiling fine-grained SPMD-threaded code (e.g., GPU CUDA code) for multicore CPUs. We point out some correctness pitfalls in existing techniques, particularly in their treatment to implicit synchronizations. We then describe a systematic dependence analysis specially designed for handling implicit synchronizations in SPMD-threaded programs. By unveiling the relations between inter-thread data dependences and correct treatment to synchronizations, we present a dependence-based solution to the problem. Experiments demonstrate that the proposed techniques can resolve the correctness issues in existing compilation techniques, and help compilers produce correct and efficient translation results. Details have been published at PACT 2011.

In the second part, we focus on some efficiency issues in the code generation during the translation. Due to the architectural differences among different types of devices, some coding tradeoff suitable for exploiting one type of processors may not fit another well. An SPMD-translation that is oblivious to such differences may end up producing code of inferior efficiency. In this work, we concentrate on two main sources of inefficiency that limits existing GPU-to-CPU translators.

The first relates with synchronizations in GPU programs. A GPU kernel is usually executed simultaneously by thousands of threads. Fine-grained synchronizations (e.g., locks) are rarely used. They are especially error-prone at such a large scale of parallelism. And furthermore, they often require the insertion of some thread-ID–based conditional statements for distinguishing threads with different synchronization needs. These statements often hurt GPU performance substantially due to the weakness of GPU in handling conditional branches. As a result, many GPU programs employ barriers, causing *overly strong synchronizations* whose constraints are stronger than necessary. In this work, we sometimes call these synchronizations *relaxable synchronizations*.

The second source of inefficiency is in the redundant computations in GPU programs. In GPU programs, there are typically more redundant computations than in CPU programs: As GPU is strong in supporting massive parallelism but weak in handling conditional branches, it is common for a GPU program to allow some threads to carry some useless computations because otherwise, some conditional branches would have to be introduced. In the parallel reduction code in CUDA SDK, for instance, more than half of all threads conduct some useless computations. Such redundancies are often acceptable on GPU as they overlap with useful calculations for the massive parallelism of GPU. But when getting into the translated CPU code, they may cause serious inefficiency. In the case of parallel reduction, the useless computations in the execution of the translated CPU code weights more than half of the entire execution time. Such a Redundancy differs from the traditional concept of redundancy in that they are thread-dependent. We call it *thread partial redundancy*.

In this work, we propose a unified solution to both issues. The key insight is that the two kinds of inefficiency essentially come from a single reason: the non-uniformity among GPU threads. We develop a thread-level fine-grained analysis framework to address both types of inefficiency. Unlike existing GPU-to-CPU translations, the analysis target of our framework is not the static instructions but their instances executed by each thread. The framework uses *thread-level dependence graphs (TLDG)* to model the relations among the dynamic instances of GPU instructions in the executions of different threads. TLDG has a bounded size, with edges capturing critical dependences. Based on the TLDG, we develop a CPU code generator that feature an instance-level instruction scheduler, the use of graph pattern matching and instance-level conditional branch elimination for the optimization of the generated code, and a scheme for identifying thread partial redundancy from the TLDG and preventing them from getting into the generated CPU program. The techniques are able to address both types of inefficiency effectively, improving the performance of the produced CPU program by as much as a factor of four. This work has been published at ICS 2012.

2.5 Energy Efficiency through Hybrid Memory Systems on GPU

Hybrid memory designs, such as DRAM plus Phase Change Memory (PCM), have shown some promise for alleviating power and density issues faced by traditional memory systems. But previous studies have concentrated on CPU systems with a modest level of parallelism. In this work, we studied the problem in a massively parallel setting. Specifically, we investigated the special implications to hybrid memory imposed by the massive parallelism in GPU. The study empirically shows that, contrary to promising results demonstrated for CPU, previous designs of PCM-based hybrid memory result in significant degradation to the energy efficiency of GPU. It reveals that the fundamental reason comes from a multi-facet mismatch between those designs and the massive parallelism in GPU. It presents a solution that centers around a close cooperation between compiler-directed data placement and hardware-assisted runtime adaptation. On the hardware side, we designed a novel hybrid memory system with runtime data migration support as shown in Figure 3. On the software side, we proposed a new way to estimate the appropriate data placement on hybrid memory through dynamic programming on a data placement cost graph as illustrate in Figure 4. The co-design approach helps tap into the full potential of hybrid memory for GPU without requiring dramatic hardware changes over previous designs, yielding 6% and 49% energy saving on average compared to pure DRAM and pure PCM respectively, and keeping performance loss less than 2%. The results have been published at PACT 2013.

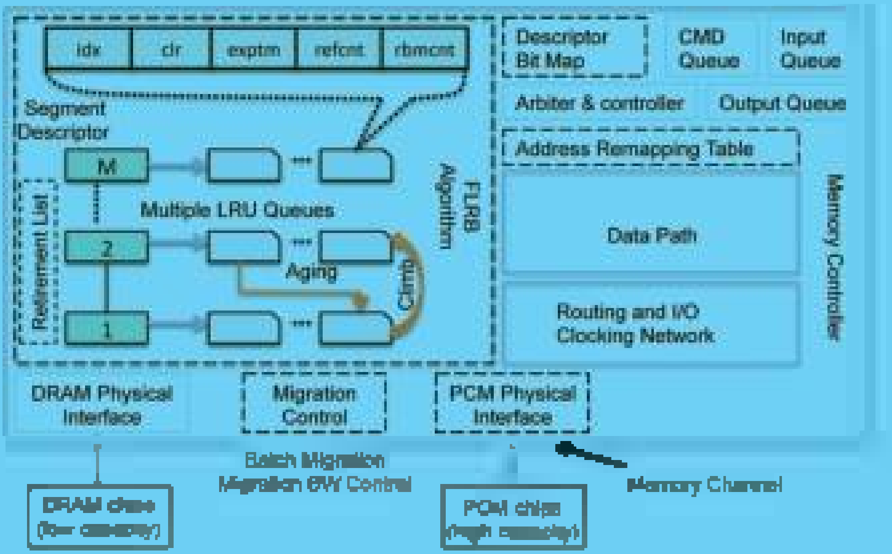

Figure 3. Memory controller with data migration logics. The dotted black boxes depict added logics.

2.6 Profile Rectification for High Performance

Sampling is a common way to do efficient profiling for code optimizations. In this work, we give a systematic exploration on the relations among sampling rates, profile accuracy, and profile usefulness for feedback-directed program optimizations (FDO). The exploration covers seven factors in four levels. It reveals some counter-intuitive relations, the most prominent of which are that higher sampling rates (within a typical sample rate range) do not lead to more accurate profiles when uniform sampling is used, and more importantly, no matter which sampling method is used, the accuracy of the profiles does not show a strong correlation with their usefulness for FDO. The study then provides a detailed analysis and points out that two types of sampling errors, *0-counter errors* and *inconsistent branch taken rate errors*, play an essential role in restraining the power of FDO. Based on empirically confirmed cross-input stableness of two kinds of value patterns in profiles, we present a simple way to rectify the two types of errors through a training profile. The dramatic enhancement of the FDO benefits concludes that simple rectification of the two types of errors in a profile is promising in tapping into the full potential of FDO. It also suggests that with profile rectification, sampling rate (and hence sample overhead) can be significantly lowered without hurting the FDO benefits.

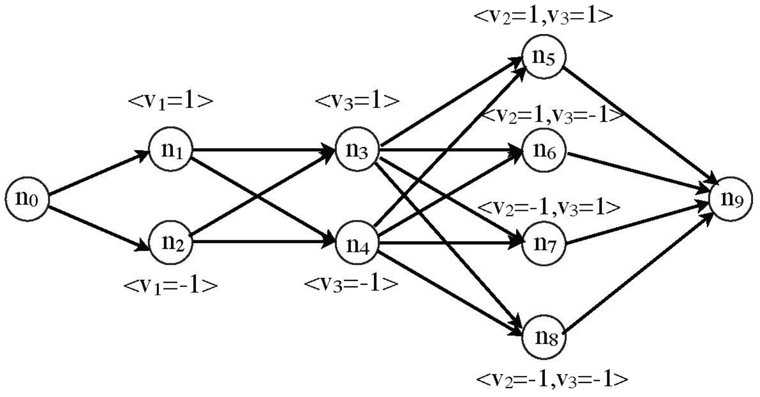

Figure 4: A Placement Cost Graph of a program with three kernels. Each column of nodes represents the possible ways to place the arrays used in one GPU kernel in the program. Each node represents a scheme to place the arrays. Each edge has a weight equaling the cost of data migration from one scheme to another (the weights are not shown for readability.)

These multi-fold novel insights provide the first principled understanding in effective collection of profiles for FDO. They may help advance the profile collection in modern runtime systems, and open up many new opportunities for modern program optimizations. This work has been published on the European Conference on Object-Oriented Programming (ECOOP'2013).

2.7 Enabling Flexible Control of Scheduling on GPU

Recent years have seen increasing popularity of Graphics Processing Units (GPU) in general-purpose computing, thanks to massive parallelism provided by GPU. With hundreds of cores integrated, GPU often creates tens of thousands of threads for an application. The massive parallelism produces large potential throughput, but also imposes grand challenges for thread management, or scheduling.

Scheduling determines when and where a task is processed. It is essential for matching communication and memory access patterns with underlying architecture, in order to fully tap into the power of a parallel system. Scheduling is usually controlled by thread schedulers. On CPU, the thread scheduling is implemented through system APIs. But on GPU, there is no such software API; the scheduling on GPU has been controlled by hardware and runtime. Such a design is demanded by the scale of parallelism: Hundreds of thousands of threads need to be scheduled in no time. However, the lack of software-level control of scheduling forms a major barrier for software to leverage scheduling to optimize program executions. What increases the barrier is that the scheduling algorithms employed by GPU hardware and runtime have remained non-disclosed; the schedulers vary substantially across generations and have exhibited some obscure and non-deterministic behaviors.

The restrictions have drawn some recent attentions from researchers in various domains. A number of studies independently invented the method of *persistent threads* to go around the hardware scheduling problem. The idea is to create only a small number of threads that can simultaneously run actively on a GPU. Unlike in traditional kernels where a thread terminates as it finishes a task, these threads stay alive throughout the execution of a kernel function. They continuously fetch and execute tasks from one or more task queues. By controlling the order of the tasks in the queues, one can match the executions with some communication patterns among tasks—for example, putting a producer and its consumer into the same queue.

Although persistent threads offer some support to task scheduling on GPU, the support is restrictive. It only decides which tasks map to which persistent thread and their execution order; it gives no support for deciding *where or on which processor a task should run*. Such location control is still up to the hardware and proprietary runtime, which decide the placement of persistent threads, and hence the placement of tasks associated with those threads.

Lack of such scheduling control at the spatial dimension hinders persistent threads in supporting optimizations that are related with non-uniformity in processors. For instance, a modern GPU consists of multiple streaming multiprocessors (SM), with each containing tens of cores. Cores on one SM usually share some on-chip storage on that SM (e.g., L1 cache and texture cache). As a result, one task may be able to read the data in a cache brought by another task that concurrently runs on the same SM. With location control, one could make two tasks that share lots of data run concurrently on the same SM[1]. Such optimizations are especially beneficial for tasks with non-uniform data sharing, which include tasks of many irregular applications (e.g., N-body simulations), as well as tasks coming from different kernels (or applications) that are deployed concurrently on a GPU. Besides for data sharing, the spatial control is critical when there are architectural variations among SMs. Unintentional variations among SMs in a GPU already widely exist today; with frequency scaling possibly implemented in future GPU, even more substantial (intentional) variations (e.g., different SMs could be reconfigured to different clock frequencies to balance energy and performance) are possible. In these scenarios, spatial control of scheduling is important for matching tasks with the suitable SMs.

In this work, we show that spatial scheduling control actually can be enabled through a simple program transformation, called *SM-centric transformation*.

SM-centric transformation includes two essential techniques. The first is *SM-based task selection*. In a traditional GPU kernel execution, with or without persistent threads, what tasks a thread executes are usually based on the ID of the thread (or determined randomly in a dynamic task management). While with SM-based task selection, what tasks a thread executes is based on the ID of the SM that the thread runs on. By replacing the binding between tasks and threads with the binding between tasks and SMs, the scheme enables a direct, precise control of task placement on SM.

The second technique is *filling-retreating scheme*, which offers a flexible control of the amount of active threads on an SM. Importantly, the control is resilient to the randomness and obscuration in GPU hardware thread scheduling. It helps SM-centric transformation in two aspects. First, it ensures an even distribution of active threads on SMs, which is vital for guaranteeing the correctness of SM-centric transformations. Second, it facilitates online determination of the parallelism level suitable for a kernel, which is especially important for the performance of multiple-kernel co-runs, a scenario benefiting significantly from SM-centric transformation.

SM-centric transformation, by enabling flexible program-level control of task scheduling, opens up many new opportunities for optimizations. In our experiments on 72 co-runs of kernels, it helps produce on average 33% improvement in system throughput and turnaround time. When applied to locality enhancement, the enabled spatial scheduling shortens the execution times of four irregular applications by 20% on average. In both cases, it significantly outperforms the support that persistent threads provide. These results indicate that SM-centric transformation, by complementing prior methods, provides a critical missing piece of the puzzle for enabling a flexible control of task scheduling on GPU. The work has been published at ICS 2015.

2.8 Parallelizing the Inherently Sequential

Due to the Amdalh's law, even though enhancing the performance of the parallel part of a program is important, parallelizing the sequential part of an application really raises the performance limit of a parallel program. In this part of research, we concentrated on the most challenging cases for parallelization, includ-

[1] Mapping two tasks to the same persistent thread can also make them map to the same SM, but the tasks have to run serially by that thread, throttling the benefits of synergistic data fetching.

ing some inherently sequential algorithms, such as Finite State Machines (FSM).

FSM applications are important for many domains. But FSM computation is inherently sequential, making such applications notoriously difficult to parallelize. Most prior methods address the problem through speculations on simple heuristics, offering limited applicability and inconsistent speedups. In this work, we provide some principled understanding of FSM parallelization, and offer the first disciplined way to exploit application-specific information to inform speculations for parallelization. Through a series of rigorous analysis, this study develops a probabilistic model that captures the relations between speculative executions and the properties of the target FSM and its inputs. With the formulation, it proposes two model-based speculation schemes that automatically customize themselves with the suitable configurations to maximize the parallelization benefits. This rigorous treatment yields near-linear speedup on applications that state-of-the-art techniques can barely accelerate. The results have been published at the Nineteenth International Conference on Architectural Support for Programming Languages and Operating (ASPLOS'2014).

3 Student Training

Six of the doctoral students that have been involved in this project have graduated, including a female student. Upon their graduations, three of them became assistant professors at some major research universities (Rutgers University, University of California at Riverside, Colorado School of Mines), and three joined some major companies (Google, Microsoft, IBM).

4 Major Publications in the Report Period

TACO'13 "HPar: A Practical Parallel Parser for HTML", Zhijia Zhao, Michael Bebenita, Dave Herman, Jianhua Sun, Xipeng Shen, the ACM Transactions on Architecture and Code Optimization, Vol 10 Issue 4, 2013. DOI: 10.1145/2541228.2555301.

IJPP'13 "An Infrastructure for Tackling Input-Sensitivity of GPU Program Optimizations", Xipeng Shen, Yixun Liu, Eddy Z. Zhang, Poornima Bhamidipati, International Journal of Parallel Programming, vol. 41, no. 6, pages 855-869, DOI: 10.1007/s10766-012-0236-3, December, 2013.

TPDS'12 "The Significance of CMP Cache Sharing on Contemporary Multithreaded Applications" Eddy Zhang, Yunlian Jiang, Xipeng Shen, IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 2, pages 367-374, DOI: 10.1109/TPDS.2011.130, February, 2012.

TPDS'11 "The Complexity of Optimal Job Co-Scheduling on Chip Multiprocessors and Heuristics-Based Solutions", Yunlian Jiang, Kai Tian, Xipeng Shen, Jinghe Zhang, Jie Chen, Rahul Tripathi, IEEE Transactions on Parallel and Distributed Systems, Vol. 22 Issue 7, pages 1192-1205, DOI: 10.1109/TPDS.2010.193, July, 2011.

Micro'14 "PORPLE: An Extensible Optimizer for Portable Data Placement on GPU", Guoyang Chen, Bo Wu, Dong Li, Xipeng Shen, The 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, December, 2014. DOI:10.1109/MICRO.2014.20.

OOPSLA'14 "Space-Efficient Multi-Versioning for Input-Adaptive Feedback-Driven Program Optimizations", Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, Graham Yiu, SPLASH/OOPSLA, Portland, 2014. DOI:10.1145/2714064.2660229.

OOPSLA'14 "Call Sequence Prediction through Probabilistic Calling Automata", Zhijia Zhao, Bo Wu, Mingzhou Zhou, Yufei Ding, Jianhua Sun, Xipeng Shen, Youfeng WuSPALSH/OOPSLA, Portland, 2014. DOI:10.1145/2714064.2660221.

Ubicomp'14 "SatScore: Uncovering and Avoiding a Principled Pitfall in Responsiveness Measurements of App Launches", Zhijia Zhao, Mingzhou Zhou, Xipeng Shen, The 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing, Seattle, WA, 2014. DOI:10.1145/2632048.2632080.

ASE'14 "Localization of concurrency bugs using shared memory access pairs", Wenwen Wang, Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Xipeng Shen, Xiang Yuan, Jianjun Li, Xiaobing Feng, Yong Guan, the 29th IEEE/ACM International Conference on Automated Software Engineering, Sept, 2014, pages 611-622. DOI:10.1145/2642937.2642972.

ASPLOS'14 "Challenging the Embarrassingly Sequential: Parallelizing Finite State Machine-Based Computations through Principled Speculation", Zhijia Zhao, Bo Wu, Xipeng Shen, ASPLOS, Salt Lake city, 2014. DOI: 10.1145/2541940.2541989.

ASPLOS'14 "Finding the Limit: Examining the Potential and Complexity of Compilation Scheduling for JIT-Based Runtime Systems", Yufei Ding, Mingzhou Zhou, Zhijia Zhao, Sarah Eisenstat, Xipeng Shen, ASPLOS, Salt Lake City, 2014. DOI: 10.1145/2541940.2541945.

PACT'13 "Exploring Hybrid Memory for GPU Energy Efficiency through Software-Hardware Co-Design", Bin Wang, Bo Wu, Yizheng Jiao, Dong Li, Xipeng Shen, Weikuan Yu, Jeffrey Vetter, PACT, Edinburgh, Scotland, 2013. DOI: 10.1109/PACT.2013.6618807.

MSPC'13 "Software-level Scheduling to Exploit Non-uniformly Shared Data Cache on GPGPU", Bo Wu, Weilin Wang, Xipeng Shen, MSPC'13, Seattle, USA, June, 2013. DOI: 10.1145/2492408.2492421.

ECOOP'13 "Simple Profile Rectifications Go A Long Way", Bo Wu, Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, Raul Silvera, Graham Yiu, ECOOP'13, Montpellier, France, July, 2013. DOI: 10.1007/978-3-642-39038-8_27.

PPOPP'13 "Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-Coalesced Memory Accesses on GPU", Bo Wu, Zhijia Zhao, Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen, PPOPP, Shenzhen, China, 2013. DOI: 10.1145/2517327.2442523.

CGO'13 "ProfMig: A Framework for Flexible Migration of Program Profiles Across Software Versions", Mingzhou Zhou, Bo Wu, Yufei Ding, and Xipeng Shen, CGO, Shenzhen, China, 2013.
DOI: 10.1109/CGO.2013.6494984.

OOPSLA'12 "Exploiting Inter-Sequence Correlations for Program Behavior Prediction", B. Wu and Z. Zhao and X. Shen and Y. Jiang and Y. Gao and R. Silvera, SPLASH/OOPSLA, Tucson, AZ, 2012. DOI: 10.1145/2384616.2384678.

PACT'12 "Speculative Parallelization Needs Rigor: Probabilistic Analysis for Optimal Speculation of Finite State Machine Applications", Z. Zhao and B. Wu and X. Shen, PACT, Minneapolis, MN, 2012. (2-page poster paper) DOI: 10.1145/2370816.2370882.

ICS'12 "One Stone Two Birds: Synchronization Relaxation and Redundancy Removal in GPU-CPU Translation", Z. Guo, B. Wu and X. Shen, ACM International Conference on Supercomputing,Venice, Italy, 2012. DOI: 10.1145/2304576.2304583.

JSSPP'12 "Optimal Co-Scheduling to Minimize Makespan on Chip Multiprocessors", K. Tian, Y. Jiang, X. Shen and W. Mao, 16th Workshop on Job Scheduling Strategies for Parallel Processing, Shanghai, China, May, 2012, in conjunction with IPDPS'12. DOI: 10.1007/978-3-642-35867-8_7.

MSPC'12 "A Study Towards Optimal Data Layout for GPU Computing", Z. Zhang, H. Li, and X. Shen, in Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC 2012), June, 2012, in conjunction with PLDI'12. DOI: 10.1145/2247684.2247699.

PACT'11 "Enhancing Data Locality for Dynamic Simulations through Asynchronous Data Transformations and Adaptive Control", Bo Wu, Eddy Zhang, Xipeng Shen, The Twentieth International Conference on Parallel Architectures and Compilation Techniques, Galveston Island, Texas, USA, Oct, 2011. DOI: 10.1109/PACT.2011.56.

PACT'11 "Correctly Treating Synchronizations in Compiling Fine-Grained SPMD-Threaded Programs for CPU", Ziyu Guo, Eddy Zhang, Xipeng Shen, The Twentieth International Conference on Parallel Architectures and Compilation Techniques, Galveston Island, Texas, USA, Oct, 2011. DOI: 10.1109/PACT.2011.62.

LCPC'11 "Fine-Grained Treatment to Synchronizations in GPU-to-CPU Translation", Ziyu Guo, Xipeng Shen, The 24th International Workshop on Languages and Compilers for Parallel Computing, Colorado, USA, September, 2011. DOI: 10.1007/978-3-642-36036-7_12.

OOPSLA'11 "A Step Towards Transparent Integration of Input-Consciousness into Dynamic Program Optimizations", Kai Tian, Eddy Zhang, Xipeng Shen, 2011 ACM International Conference on Systems, Programming, Languages and Applications, Portland, Oregon, USA, Oct, 2011. DOI: 10.1145/2048066.2048103.