Final Report

Award number: DE-SC0013700

Recipient: The College of William and Mary (prior to July 2014) North Carolina State University (since July 2014)

Project Title: Data Locality Enhancement of Dynamic Simulations for Exascale Computing

PI: Xipeng Shen (xshen5@ncsu.edu)

Date: 11/29/2019

1 Introduction

This document summarizes the progress of the research project. It consists of three parts. The first part gives a brief review of the project background and goal, the second part reports the main progress achieved before the PI moved to NCSU, and the third part reports the progress after the move. The major publications are listed at the end of the document.

2 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 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 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.

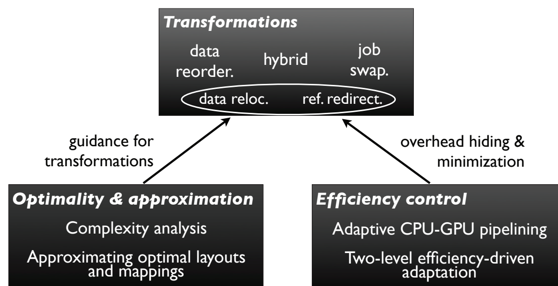

Figure 1: Major components of GStreamline.

Some 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 transformations. Our 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.

3 Progress Before Moving to NCSU (2011-2014)

3.1 GStreamline for Runtime Memory Optimizations on GPU

First, we investigated the memory reference bottleneck on GPU computing. Based on a prototype in our earlier study (ASPLOS'11), 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 **??** 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

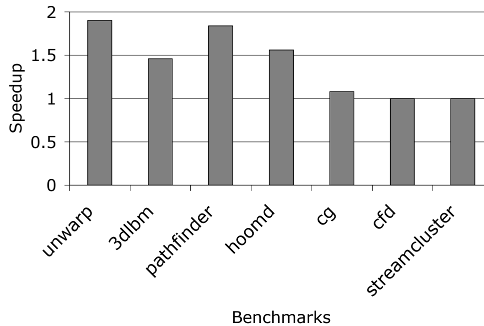

Figure 2: Speedup brought by GStreamline on NVIDIA Tesla 1060. The baseline is the performance of the original GPU programs.

challenges to the optimizations, but the runtime adaptive control successfully prevents the optimizations from causing any slowdown.

3.2 Towards Optimal Data Layout 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.

3.3 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 in a conference paper (PACT'11).

3.4 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 a conference paper (PACT11b).

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 in a conference paper (ICS12).

4 Progress After Moving to NCSU (2015-now)

This part summarizes the progress after the PI moved to NCSU[1]. Despite interruptions caused by the unsuccessful postdoc hiring, I leveraged some graduate students who are either supported by my startup funds or some fellowship from the university and still managed to achieve some decent progress of the project.

4.1 SM-Centric Task Assignment on GPU

A GPU's computing power lies in its abundant memory bandwidth and massive parallelism. However, its hardware thread schedulers, despite being able to quickly distribute computation to processors, often fail to capitalize on program characteristics effectively, achieving only a fraction of the GPU's full potential.

[1] A time gap between 2014 and 2015 was due to the grant transition delay.

Moreover, current GPUs do not allow programmers or compilers to control this thread scheduling, forfeiting important optimization opportunities at the program level. This paper presents a transformation centered on Streaming Multiprocessors (SM); this software approach to circumventing the limitations of the hardware scheduler allows flexible program-level control of scheduling. By permitting precise control of job locality on SMs, the transformation overcomes inherent limitations in prior methods.

With this technique, flexible control of GPU scheduling at the program level becomes feasible, which opens up new opportunities for GPU program optimizations. The second part of the paper explores how the new opportunities could be leveraged for GPU performance enhancement, what complexities there are, and how to address them. We show that some simple optimization techniques can enhance co-runs of multiple kernels and improve data locality of irregular applications, producing 20-33% average increase in performance, system throughput, and average turnaround time. A paper on this work has been accepted to publish at the 2015 ACM International Conference on Supercomputing.

4.2 PORPLE: An Extensible Optimizer for Portable Data Placement on GPU

GPU has complex memory systems. Where to place the data is important for the performance of a GPU program. However, the decision is difficult for a programmer to make because of architecture complexity and the sensitivity of suitable data placements to input and architecture changes.

This paper presents PORPLE, a portable data placement engine that enables a new way to solve the data placement problem. 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, computes on the fly the appropriate placement schemes for the 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 work has been published at the 47th Annual IEEE/ACM International Symposium on Microarchitecture and the Juy/August Issue, the Heterogeneous Computing special issue of IEEE Micro, 2015.

In addition, we have achieved some findings in software engagement with CPU sleep state management, space-efficient multi-versioning for input-adaptive feedback-driven program optimizations, and call sequence prediction through probabilistic calling automata. The results have been published at the 15th Workshop on Hot Topics in Operating Systems, OOPSLA'14, and other conferences, as listed at the end of this report.

4.3 Optimizing GPU Dynamic Kernel Launches

Supporting dynamic parallelism is important for GPU to benefit a broad range of applications. There are currently two fundamental ways for programs to exploit dynamic parallelism on GPU: a software-based approach with software-managed worklists, and a hardware-based approach through dynamic subkernel launches. Neither is satisfactory. The former is complicated to program and is often subject to some load imbalance; the latter suffers large runtime overhead.

In this work, we propose free launch, a new software approach to overcoming the shortcomings of both methods. It allows programmers to use subkernel launches to express dynamic parallelism. It employs a novel compiler-based code transformation named subkernel launch removal to replace the subkernel launches with the reuse of parent threads. Coupled with an adaptive task assignment mechanism, the transformation reassigns the tasks in the subkernels to the parent threads with a good load balance. The technique requires no hardware extensions, immediately deployable on existing GPUs. It keeps the programming con-

venience of the subkernel launch-based approach while avoiding its large runtime overhead. Meanwhile, its superior load balancing makes it outperform manual worklist-based techniques by 3X on average. The work was published at the 48th Annual IEEE/ACM International Symposium on Microarchitecture (Micro'2015) and IEEE Transactions on Computers (TC'2016).

4.4 Enabling Reference-Discerning Data Placement on GPU

A Graphic Processing Unit (GPU) system is typically equipped with many types of memory (e.g., global, constant, texture, shared, cache). Data placement determines what data are placed on which type of memory, essential for GPU mem- ory performance. Prior optimizations of data placement al- ways require a single view of a data object on memory, which limits the optimization effectiveness. In this work, we pro- pose coherence-free multiview, an approach that allows multi- ple views of a single data object to co-exist on GPU memory during a GPU kernel execution. We demonstrate that under certain conditions, the multiple views can remain incoherent while facilitating enhanced data placement. We present a the- orem and some compiler support to ensure the soundness of the usage of coherence-free multiview. We further develop reference-discerning data placement, a new way to enhance data placements on GPU. It enables more flexible data place- ments by using coherence-free multiview to leverage the slack in coherence requirement of some GPU programs. Experiments on three types of GPU systems show that, with less than 200KB space cost, the new data placement technique can pro- vide a 1.6X average (up to 4.27X) speedup. This work has been published at the ACM International Conference on Supercomputing (ICS'16).

4.5 Algorithm Optimizations for Data Analytics

In addition, we have examined algorithmic optimizations to data analytics problems, especially those that involve lots of distance calculations. Computing distances among data points is an essential part of many important algorithms in data analytics, graph analysis, and other domains. In each of these domains, developers have spent significant manual effort optimizing algorithms, often through novel applications of the triangle equality, in order to minimize the number of distance computations in the algorithms. In this work, we observe that many algorithms across these domains can be generalized as an instance of a generic distance-related abstraction. Based on this abstraction, we derive seven principles for correctly applying the triangular inequality to optimize distance-related algorithms. Guided by the findings, we develop Triangular OPtimizer (TOP), the first software framework that is able to automatically produce optimized algorithms that either matches or outperforms manually designed algorithms for solving distance-related problems. TOP achieves up to 237x speedups and 2.5X on average on CPU, and 8X speedups on GPU. The work has been published at the 32nd International Conference on Machine Learning (ICML'15), the 41st International Conference on Very Large Data Bases (VLDB'15).

4.6 Software Framework for Enabling Efcient Scheduling of GPU

Since last year, we started investigating thread scheduling optimizations on GPU to further maximize the computing efficiency. We have achieved some preliminary progress, especially on creating software controllability of the schedule of multiple kernels, published at PPoPP'17. There are still some important potential for translating the software controllability to optimizations of program executions, which we plan to explore in the coming year if we could get the no cost extension.

5 Major Publications in the Report Period

Publications after moving to NCSU are in bold font.

[PPoPP'17] "EfSha: A Software Framework for Enabling Efcient Preemptive Scheduling of GPU", G. Chen, Y. Zhao, X. Shen, H. Zhou, the 22nd ACM SIGPLAN Symposium on Principles and Practice

of Parallel Programming, Feb 2017.

[TC'16] "Optimizing Data Placement on GPU Memory: A Portable Approach", G. Chen, X. Shen, B. Wu, D. Li, IEEE Transactions on Computers, DOI: 10.1109/TC.2016.2604372, 2016.

[ICS'16] "Coherence-Free Multiview: Enabling Reference-Discerning Data Placement on GPU", Guoyang Chen, Xipeng Shen, ACM International Conference on Supercomputing, 2016.

[PLDI'15] "Autotuning Algorithmic Choice for Input Sensitivity", Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, Saman Amarasinghe, the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation, Portland, Orgon, June 13-17, 2015. (19% acceptance rate)

ICS'15 "Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations", Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, Jeffrey Vetter, ACM International Conference on Supercomputing, Newport Beach, CA, 2015. (25% acceptance rate)

HotOS'15 "Software Engagement with Sleeping CPUs", Qi Zhu, Meng Zhu, Bo, Wu, Xipeng Shen, Kai Shen, Zhiying Wang, the 15th Workshop on Hot Topics in Operating Systems, Kautause Ittigen, Switzerland, May, 2015. (32% acceptance rate)

IEEE/Micro'15 "Enabling Portable Optimizations of Data Placement on GPU", Guoyang Chen, Bo Wu, Dong Li, Xipeng Shen, Juy/August Issue, the Heterogeneous Computing special issue of IEEE Micro, 2015.

ASPLOS'15 "On-the-Fly Principled Speculation for FSM Parallelization", Zhijia Zhao, Xipeng Shen, The 20th International Conference on Architectural Support for Programming Languages and Operating Systems, Istanbul, Turkey, March 14-18, 2015. (17% acceptance rate)

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. (19% acceptance rate)

OOPSLA'14a "Space-Efficient Multi-Versioning for Input-Adaptive Feedback-Driven Program Optimizations", Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, Graham Yiu, SPLASH/OOPSLA, Portland, 2014. (28% acceptance rate)

OOPSLA'14b "Call Sequence Prediction through Probabilistic Calling Automata", Zhijia Zhao, Bo Wu, Mingzhou Zhou, Yufei Ding, Jianhua Sun, Xipeng Shen, Youfeng Wu, SPALSH/OOPSLA, Portland, 2014. (28% acceptance rate)

[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. (17

[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. (18[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. (28

[ICS'12] "One Stone Two Birds: Synchronization Relaxation and Redundancy Removal in GPU-CPU Translation", Z. Guo and B. Wu and X. Shen, ACM International Conference on Supercomputing,Venice, Italy, 2012. (22

[TPDS'11] "The Significance of CMP Cache Sharing on Contemporary Multithreaded Applications", Eddy Zhang, Yunlian Jiang, Xipeng Shen, IEEE Transactions on Parallel and Distributed Systems.

[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. Acceptance rate: 16

[PACT'11b] "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. Acceptance rate: 16

[ASPLOS'11] "On-the-Fly Elimination of Dynamic Irregularities for GPU Computing", Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, Xipeng Shen, the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems , Newport Beach, California, USA, March, 2011. Acceptance rate: 21