Report from Dagstuhl Seminar 13142

Correct and Efficient Accelerator Programming

Edited by

Albert Cohen[1], Alastair F. Donaldson[2], Marieke Huisman[3], and Joost-Pieter Katoen[4]

1 **ENS – Paris, FR,** Albert.Cohen@inria.fr
2 **Imperial College London, GB,** alastair.donaldson@imperial.ac.uk
3 **University of Twente, NL,** Marieke.Huisman@ewi.utwente.nl
4 **RWTH Aachen University, DE,** katoen@cs.rwth-aachen.de

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 13142 "Correct and Efficient Accelerator Programming". The aim of this Dagstuhl seminar was to bring together researchers from various sub-disciplines of computer science to brainstorm and discuss the theoretical foundations, design and implementation of techniques and tools for correct and efficient accelerator programming.

Seminar 1.–4. April, 2013 – www.dagstuhl.de/13142
1998 ACM Subject Classification C.1.4 Processor Architectures: Parallel Architectures, D.1.3 Programming Techniques: Concurrent Programming, D.3.4 Programming Languages: Processors – Compilers, Optimization, F.3.1 Logics and Meanings of Programs: Specifying and Verifying and Reasoning about Programs
Keywords and phrases Accelerator programming, GPUs, Concurrency, Formal verification, Compilers, Memory models, Architecture, Parallel programming models
Digital Object Identifier 10.4230/DagRep.3.4.17
Edited in cooperation with Jeroen Ketema

1 Executive Summary

Albert Cohen
Alastair F. Donaldson
Marieke Huisman
Joost-Pieter Katoen

License Creative Commons BY 3.0 Unported license
© Albert Cohen, Alastair F. Donaldson, Marieke Huisman, and Joost-Pieter Katoen

In recent years, massively parallel accelerator processors, primarily GPUs, have become widely available to end-users. Accelerators offer tremendous compute power at a low cost, and tasks such as media processing, simulation and eye-tracking can be accelerated to beat CPU performance by orders of magnitude. Performance is gained in energy efficiency and execution speed, allowing intensive media processing software to run in low-power consumer devices. Accelerators present a serious challenge for software developers. A system may contain one or more of the plethora of accelerators on the market, with many more products anticipated in the immediate future. Applications must exhibit portable correctness, operating correctly on any configuration of accelerators, and portable performance, exploiting processing power and energy efficiency offered by a wide range of devices.

Except where otherwise noted, content of this report is licensed
under a Creative Commons BY 3.0 Unported license
Correct and Efficient Accelerator Programming, *Dagstuhl Reports*, Vol. 3, Issue 4, pp. 17–33
Editors: Albert Cohen, Alastair F. Donaldson, Marieke Huisman, and Joost-Pieter Katoen

Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The seminar focussed on the following areas:

- Novel and attractive methods for constructing system-independent accelerator programs;
- Advanced code generation techniques to produce highly optimised system-specific code from system-independent programs;
- Scalable static techniques for analysing system-independent and system-specific accelerator programs both qualitatively and quantitatively.

The seminar featured five tutorials providing an overview of the landscape of accelerator programming:

- Architecture – Anton Lokhmotov, ARM
- Programming models – Lee Howes, AMD
- Compilation techniques – Sebastian Hack, Saarland University
- Verification – Ganesh Gopalakrishnan, University of Utah
- Memory models – Jade Alglave, University College London

In addition, there were short presentations from 12 participants describing recent results or work-in-progress in these areas, and two discussion sessions:

- Domain specific languages for accelerators;
- Verification techniques for GPU-accelerated software.

Due to the "correctness" aspect of this seminar, there was significant overlap of interest with a full week seminar on *Formal Verification of Distributed Algorithms* running in parallel. To take advantage of this overlap a joint session was organised, featuring a talk on verification of GPU kernels by Alastair Donaldson, Imperial College London (on behalf of the *Correct and Efficient Accelerator Programming* seminar) and a talk on GPU-accelerated runtime verification by Borzoo Bonakdarpour, University of Waterloo, on behalf of the *Formal Verification of Distributed Algorithms* seminar.

2 Table of Contents

3 Overview of Talks

3.1 Weak Memory Models: A Tutorial

Jade Alglave (University College London, GB)

License Creative Commons BY 3.0 Unported license
© Jade Alglave

In this talk I presented several behaviours observed on current processors such as Intel x86, IBM Power and ARM. These behaviours demonstrate that one cannot assume Sequential Consistency (SC) to model executions of concurrent programs.

I also explained which synchronisation one should use to enforce an SC behaviour on these examples. I concluded with an excerpt of the PostgreSQL database software which features two idioms that do not behave in an SC manner if not synchronised properly.

3.2 Estimating the WCET of GPU-Accelerated Applications using Hybrid Analysis

Adam Betts (Imperial College London, GB)

License Creative Commons BY 3.0 Unported license
© Adam Betts
Joint work of Betts, Adam; Donaldson, Alastair F.

Gaining good performance from applications running on GPUs remains very challenging. One approach is to generate several kernel variants and then choose the best among this set using performance analysis techniques. Most approaches to performance analysis focus on average-case behaviour, but this sometimes yields ties between several kernels. In this talk, we present work performed in the context of the CARP project which focuses on estimating an outlier execution time, namely the Worst-Case Execution Time (WCET), in order to break ties and to estimate performance from a worst-case perspective. The technique we present is based on a mixture of dynamic and static analyses, which operates by collecting execution times of small program segments from measurements and then combining the data using a static program model. The talk focuses on extensions needed to incorporate concurrency into the timing model, in particular how to model the stalls experienced by warps (groups of threads on NVIDIA hardware) while other warps are in flight. We present our tool to estimate the WCET of GPU kernels, which is based on GPGPU-sim, and results when analysing several benchmarks from the CUDA SDK.

This work was supported by the EU FP7 STREP project CARP.

3.3 GPUVerify: A Verifier for GPU Kernels

Alastair F. Donaldson (Imperial College London, GB)

License Creative Commons BY 3.0 Unported license
© Alastair F. Donaldson
Joint work of Betts, Adam; Chong, Nathan; Collingborne, Peter; Donaldson, Alastair F.; Ketema, Jeroen; Kyshtymov, Egor; Qadeer, Shaz; Thomson, Paul;
Main reference A. Betts, N. Chong, A.F. Donaldson, S. Qadeer, P. Thomson, "GPUVerify: a Verifier for GPU Kernels," in Proc. of the 27th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12), pp. 113–132, ACM, 2012.
URL http://dx.doi.org/10.1145/2398857.2384625

The motivation for general-purpose computing on using graphics processing units (GPUs) is *performance*, thus GPU programmers work hard to write highly-optimised low-level kernels in OpenCL and CUDA. It is hard to write such optimised code correctly, thus there is wide scope for bugs, especially data races, in GPU kernels.

In this talk I presented GPUVerify, a technique and tool for verifying race-freedom of GPU kernels. GPUVerify performs analysis of GPU kernels by reducing the verification task to that of verifying a sequential program. This is achieved by exploiting (a) the fact that data race-freedom is a *pairwise* property, allowing a reduction to an arbitrary pair of threads, and (b) the observation that as long as data races are not tolerated, it is sufficient to consider a single, canonical schedule between pairs of barriers. Combined with source-level predicated execution, this allows a GPU kernel to be transformed into a sequential program that models round-robin execution of a pair of threads, equipped with instrumentation to perform data race detection. Proving data race freedom then amounts to proving correctness of this sequential program; in GPUVerify this is achieved by reusing the Boogie verification system.

During the talk I gave a demo of GPUVerify, and also illustrated manually how the automatic translation into Boogie is achieved.

This work was supported by the EU FP7 STREP project CARP.

3.4 Bulk Synchronous Streaming Model for the MPPA-256 Manycore Processor

Benoît Dupont de Dinechin (Kalray – Orsay, FR)

License Creative Commons BY 3.0 Unported license
© Benoît Dupont de Dinechin

The Kalray MPPA-256 is an integrated manycore processor manufactured in 28nm CMOS technology that consumes about 10W for 230GFLOPS at 400MHz. Its 256 data processing cores and 32 system cores are distributed across 16 shared-memory clusters and 4 I/O subsystems, themselves connected by two networks-on-chip (NoCs). Each Kalray core implements a general-purpose Very Long Instruction Word (VLIW) architecture with 32-bit addresses, a 32-bit/64-bit floating-point unit, and a memory management unit.

This talk explains the motivations and the directions for the development of a streaming programming model for the MPPA-256 processor. Like the IBM Cell/BE or the Intel SCC, the Kalray MPPA-256 architecture is based on clusters of general-purpose cores that share a local memory, where remote memory accesses require explicit communication. By comparison, GP-GPU architectures allow direct access to the global memory and hide the resulting latency with massive hardware multithreading. On-going port of OpenCL to the MPPA-256 processor

may only reach limited performances and gives up run-time predictability, as the global memory has to be emulated by software with a Distributed Shared Memory (DSM) technique.

The alternative we propose is to develop a stream-oriented programming model called 'Bulk Synchronous Streaming' (BSS), by adapting the classic Bulk Synchronous Parallel (BSP) model. The BSP model belongs to the family of symmetric parallel programming models for distributed memory supercomputers, like Cray SHMEM and Co-Array Fortran. The adaptations envisioned for the BSS include: maintaining the global data objects in the DDR memory, instead of distributing them across the local memories; enabling execution of BSP-like programs with a number of processing images larger than the number of clusters, by streaming their execution onto the available clusters; extending the precise BSP performance model to the BSS model. The result can be characterized as a generalized vector execution model, since the global data updates are not visible until after the superstep synchronizations.

3.5 Analysis of Shared Memory and Message Passing Parallel Programs

Ganesh L. Gopalakrishnan (University of Utah, US)

License Creative Commons BY 3.0 Unported license
© Ganesh L. Gopalakrishnan

In this tutorial, a general introduction to shared memory and distributed memory programming is provided. It is important to have well-structured concurrent systems so that their debugging becomes easier. After a discussion of the general nature of large-scale computational frameworks such as Utah's Uintah system, the discussion shifts to the specifics of GPU programming. The University of Utah GPU formal correctness checking tool GKLEE. Various issues pertaining to the correctness of GPU programs are discussed including race detection, resource utilization checking using symbolic methods, and finally memory consistency issues and floating-point accuracy. A discussion of message passing verification around the ISP tool then follows. The tutorial ends with a discussion of how to make tangible impact by lowing expectations in a productive way: pick problems such as determinism and reproducibility if overall correctness is too difficult to achieve, or to even state clearly.

3.6 Compilation Techniques for Accelerators Tutorial

Sebastian Hack (Universität des Saarlandes, DE)

License Creative Commons BY 3.0 Unported license
© Sebastian Hack

I gave an overview over four compilation techniques that are relevant to accelerators.

Type-based vectorization uses the type system to guide the compiler's vectorization choices. This way, SIMD programming can be done in a portable yet efficient way because the programmer is relieved of using intrinsics or other half-hearted language extensions.

Another approach uses abstract interpretation to decide which parts of a kernel have to be vectorized or not. This is important for accelerators that use explicit SIMD instruction sets. On such machines, retaining scalar computations is essential to mediate the overhead of vectorization.

The polyhedral model uses polyhedra to represent loop nests of a certain kind. Using integer linear programming techniques, many existing loop transformations can be rephrased

to the problem of finding an affine schedule of a dependence graph annotated with dependence polyhedra and iteration space polyhedra.

Finally, I briefly spoke about the rapid development of domain-specific languages (DSLs) using language virtualization. Here, one embeds the DSL in a host language by hijacking its syntactic analysis. On top of that a custom compiler infrastructure is being built which then generates code for accelerators.

3.7 Accelerator Programming Models

Lee Howes (AMD – Sunnyvale, US)

License Creative Commons BY 3.0 Unported license
© Lee Howes
Main reference B.R. Gaster, L. Howes, "Can GPGPU Programming be Liberated from the Data Parallel Bottleneck?" IEEE Computer, 45(8):42–52, IEEE, 2012.
URL http://dx.doi.org/10.1109/MC.2012.257

Accelerator devices have become increasingly capable over time and will continue to do so. With increasing flexibility of scheduling, shared virtual memory across SoCs, device context switching and more there is a much wider range of programming models that such devices will be able to support. In this talk we discuss some of the limitations of current accelerator programming models that have arisen due to limitations in hardware capabilities or early design decisions, and some ways they may evolve to become more flexible. If time allows we will also look at some of the current leading edge models to give a broader view of what is currently available.

3.8 Specification and Verification of GPGPU Programs using Permission-Based Separation Logic

Marieke Huisman (University of Twente, NL)

License Creative Commons BY 3.0 Unported license
© Marieke Huisman
Joint work of Huisman, Marieke; Mihelčić, Matej

Graphics Processing Units (GPUs) are increasingly used for general-purpose applications because of their low price, energy efficiency and computing power. Considering the importance of GPU applications, it is vital that the behaviour of GPU programs can be specified and proven correct formally. This talk presents our ideas how to verify GPU programs written in OpenCL, a platform-independent low-level programming language. Our verification approach is modular, based on permission-based separation logic. We present the main ingredients of our logic, and illustrate its use on several example kernels. We show in particular how the logic is used to prove data-race- freedom and functional correctness of GPU applications.

This work was supported by the EU FP7 STREP project CARP.

3.9 Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels

Jeroen Ketema (Imperial College London, GB)

License Creative Commons BY 3.0 Unported license
© Jeroen Ketema
Joint work of Collingbourne, Peter; Donaldson, Alastair F.; Ketema, Jeroen; Qadeer, Shaz
Main reference P. Collingbourne, A.F. Donaldson, J. Ketema, S. Qadeer, "Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels," in Proc. of the 22nd European Symp. on Programming (ESOP 2013), LNCS, Vol. 7792, pp. 270–289, Springer, 2013.
URL http://dx.doi.org/10.1007/978-3-642-37036-6_16

In this talk I present a semantics of GPU kernels the parallel programs that run on Graphics Processing Units (GPUs). We provide a novel lock-step execution semantics for GPU kernels represented by arbitrary reducible control flow graphs and compare this semantics with a traditional interleaving semantics. We show for terminating kernels that either both semantics compute identical results or both behave erroneously.

The result induces a method that allows GPU kernels with arbitrary reducible control flow graphs to be verified via transformation to a sequential program that employs predicated execution. We implemented this method in the GPUVerify tool and experimentally evaluated it by comparing the tool with the previous version of the tool based on a similar method for structured programs, i.e., where control is organised using if and while statements.

This work was supported by the EU FP7 STREP project CARP.

3.10 On the Correctness of the SIMT Execution Model of GPUs

Alexander Knapp (Universität Augsburg, DE)

License Creative Commons BY 3.0 Unported license
© Alexander Knapp
Joint work of Knapp, Alexander; Habermaier, Axel; Ernst, Gidon
Main reference A. Habermaier, A. Knapp, "On the Correctness of the SIMT Execution Model of GPUs," in Proc. of the 21st European Symposium on Programming Languages and Systems (ESOP'12), LNCS, Vol. 7211, pp. 316–335, Springer, 2012.
URL http://dx.doi.org/10.1007/978-3-642-28869-2_16

GPUs use a single instruction, multiple threads (SIMT) execution model that executes batches of threads in lockstep. If the control flow of threads within the same batch diverges, the different execution paths are scheduled sequentially; once the control flows reconverge, all threads are executed in lockstep again. Several thread batching mechanisms have been proposed, albeit without establishing their semantic validity or their scheduling properties. To increase the level of confidence in the correctness of GPU-accelerated programs, we formalize the SIMT execution model for a stack-based reconvergence mechanism in an operational semantics and prove its correctness by constructing a simulation between the SIMT semantics and a standard interleaved multi-thread semantics. We discuss an implementation of the semantics in the K framework and a formalization of the correctness proof in the theorem prover KIV. We also demonstrate that the SIMT execution model produces unfair schedules in some cases.

3.11 Using early CARP technology to implement BLAS on the Mali GPU series

Alexey Kravets (ARM Ltd. – Cambridge, GB)

License Creative Commons BY 3.0 Unported license
© Alexey Kravets
Joint work of Kravets, Alexey; van Haastregt, Sven; Lokhmotov, Anton

We have demonstrated how early CARP technology can implement BLAS (Basic Linear Algebra Subprograms) on accelerators. We presented a high-level (DSL) description of a BLAS subroutine and PENCIL (Platform-Neutral Compute Intermediate Language) code for this subroutine. Finally we showed how OpenCL code could be generated through the DSL-PENCIL workflow.

This work was supported by the EU FP7 STREP project CARP.

3.12 Accelerator architectures and programming

Anton Lokhmotov (ARM Ltd. – Cambridge, GB)

License Creative Commons BY 3.0 Unported license
© Anton Lokhmotov

Special-purpose processors can outperform general-purpose processors by orders of magnitude, importantly, in terms of energy efficiency as well as execution speed. This talk overviews the key architectural techniques used in parallel programmable accelerators such as GPUs: vector processing and fine-grained multithreading, and challenges associated with correct and efficient accelerator programming.

3.13 Efficient Code Generation using Algorithmic Skeletons and Algorithmic Species

Cedric Nugteren (TU Eindhoven, NL)

License Creative Commons BY 3.0 Unported license
© Cedric Nugteren
Joint work of Nugteren, Cedric; Corporaal, Henk
Main reference C. Nugteren, P. Custers, H. Corporaal, "Algorithmic species: A classification of affine loop nests for parallel programming," ACM Transactions on Architecture and Code Optimization (TACO), 9(4), 40 pp., ACM, 2013.
URL http://dx.doi.org/10.1145/2400682.2400699

We presented a technique to fully automatically generate efficient and readable code for parallel processors. We base our approach on skeleton-based compilation and 'algorithmic species', an algorithm classification of program code. We use a tool to automatically annotate C code with species information where possible. The annotated program code is subsequently fed into the skeleton-based source-to-source compiler 'Bones', which generates OpenMP, OpenCL or CUDA code. This results in a unique approach, integrating a skeleton-based compiler for the first time into an automated compilation flow. We demonstrated the benefits of our approach using the PolyBench suite by presenting average speed-ups of 1.4x and 1.6x for CUDA kernel code compared to PPCG and Par4All, two state-of-the-art polyhedral compilers.

3.14 Formal Analysis of GPU Programs with Atomics via Conflict-Directed Delay-Bounding

Zvonimir Rakamarić (University of Utah, US)

License Creative Commons BY 3.0 Unported license
© Zvonimir Rakamarić
Joint work of Chiang, Wei-Fan; Gopalakrishnan, Ganesh; Li, Guodong; Rakamarić, Zvonimir
Main reference W.-F. Chiang, G. Gopalakrishnan, G. Li, Z. Rakamaric, "Formal Analysis of GPU Programs with Atomics via Conflict-Directed Delay-Bounding," in Proc. of the 5th NASA Formal Methods Symposium (NFM'13), LNCS, Vol. 7871, pp. 213–228, Springer, 2013.
URL http://dx.doi.org/10.1007/978-3-642-38088-4_15

GPU based computing has made significant strides in recent years. Unfortunately, GPU program optimizations can introduce subtle concurrency errors, and so incisive formal bug-hunting methods are essential. This paper presents a new formal bug-hunting method for GPU programs that combine barriers and atomics. We present an algorithm called Conflict-directed Delay-bounded scheduling algorithm (CD) that exploits the occurrence of conflicts among atomic synchronization commands to trigger the generation of alternate schedules; these alternate schedules are executed in a delay-bounded manner. We formally describe CD, and present two correctness checking methods, one based on final state comparison, and the other on user assertions. We evaluate our implementation on realistic GPU benchmarks, with encouraging results.

3.15 Code Generation for GPU Accelerators in the Domain of Image Preprocessing

Oliver Reiche (Universität Erlangen-Nürnberg, DE)

License Creative Commons BY 3.0 Unported license
© Oliver Reiche
Joint work of Reiche, Oliver; Membarth, Richard; Hannig, Frank; Teich, Jürgen

This talk presented the Heterogeneous Image Processing Acceleration (HIPAcc) Framework that allows automatic code generation for algorithms from the domain of image preprocessing on GPU accelerators. By decoupling the algorithm from its schedule in a domain-specific language (DSL), efficient code can be generated that leverages the computational power of such accelerators. The decoupling allows to map the algorithm to the deep memory hierarchy found in today's GPUs based on domain knowledge and an architecture model of the target machine. Based on the same algorithm description, tailored code variants can be generated for different target architectures, improving programmer productivity significantly. The generated low-level CUDA, OpenCL and Renderscript codes allow to achieve competitive performance on GPU accelerators from NVIDIA, AMD and ARM compared to hand written codes while preserving high productivity and portability.

3.16 Compositional Analysis of Concurrent Timed Systems with Horn Clauses and Interpolants (work in progress)

Philipp Rümmer (Uppsala University, SE)

License Creative Commons BY 3.0 Unported license
© Philipp Rümmer
Joint work of Hojjat, Hossein; Kuncak, Viktor; Rümmer, Philipp; Subotic, Pavle; Yi, Wang

Timed automata are a well established theory for modelling and verifying real-time systems, with many applications both in industrial and academic context. Although model checking of timed automata has been studied extensively during the last two decades, scalability of tools for analysing timed automata remains a concern, in particular when applied to instances of industrial size. When verifying networks of (concurrent) timed automata, the size of the combined state space can be a limiting factor. In this paper we present an interpolation-based predicate abstraction framework which encodes timed automata as sets of Horn clauses, with the help of Owicki-Gries and Rely-Guarantee encoding schemes.

3.17 Performance Portability Investigations for OpenCL

Ana Lucia Varbanescu (TU Delft, NL)

License Creative Commons BY 3.0 Unported license
© Ana Lucia Varbanescu
Joint work of Fang, Jianbin; Shen, Jie; Varbanescu, Ana Lucia

Multi- and many-core processors have become ubiquitous, as we — as software designers and developers — find them in all modern devices, from mobile phones to desktops, servers, and supercomputers. Their goal is to accelerate software applications, sometimes with additional constraints (like low power consumption or real-time guarantees).

To program these platforms, the users often use native, hardware-centric programming models, where applications are mapped directly in a platform-friendly form and mapped in unintuitive ways on the platform. We argue that accelerator programming needs tools that can raise the programmability (where programmability is a combination of performance, portability, and productivity). One such tool is OpenCL, a programming model aiming to tackle the problem of accelerator programming in a standardized way: it exposes a virtual computing platform to allow for functional portability, uses platform-specific backends to allow for high performance, and it provides a higher-level model of computation to help productivity.

In this talk, we present studies for three different performance portability aspects of OpenCL.

First, we compare the performance of OpenCL to that of alternative languages. When comparing OpenCL agains CUDA, we find that the results are very similar: 8 out of 12 applications have less than 10% difference in performance when programmed in the two different models. The remaining applications exhibit either compiler or programmer corner-cases, which we are able to solve and therefore match the performance of the two languages. When comparing OpenCL with OpenMP for multi-core CPUs, we expect to find much larger differences in performance. This is not always the case. In fact, about a third of the benchmarks we have used (self-made and from the Rodinia benchmark) perform better in OpenCL, the others being similar or better in OpenMP. In other words, OpenCL codes can perform well on multi-core CPUs. We are also able to show that when the performance

diverges significantly in favor of OpenMP, the causes are in the GPU-friendly optimizations that have been applied to the OpenCL code. Finally, we show how GPU OpenCL codes can be ported systematically to CPUs, thus gaining a parallel implementation with definite performance gain, only at the cost of systematic, pseudo-standard code changes. Specifically, the GPU-friendly optimizations that need undoing are: extreme fine-grained tasks, memory access patterns for coalescing, local memory usage, and vectorization. We show that these changes can be done in the form of code specialization, and do not affect the core of the 'naive' implementation. Thus, we conclude that OpenCL can be performance portable from GPUs to CPUs, and the degree of this portability depends on code specialization.

Second, we zoom in on one of the performance gaps between the CPU and GPU versions of OpenCL: local memory. In theory, using the OpenCL local memory for GPUs is considered an optimization, while using it on the CPUs is considered a mistake. However, in our extensive experimental work, we have seen a large variation in the behavior of applications that use local memory. Therefore, we considered a microbenchmarking approach, where we tested a whole set (over 40!) memory access patterns and their impact on the performance of local memory for GPUs and CPUs. The results are stored into a performance database. Given that we are able to formalize the memory access patters, we are then able to query this database and find out the predicted impact of using local memory for various applications, and decide whether applying this optimization will pay off or not. We conclude that such an approach is a viable alternative for performance prediction for local memory behavior.

Third, we present our contribution in combining the strength of OpenCL on both GPUs and CPUs: Glinda. Our Glinda framework allows (pseudo-)automated workload distribution of imbalanced applications on heterogeneous platforms, thus combining (only when needed) the execution of the same application on both CPUs and GPUs, concurrently. We briefly discuss the modules of Glinda, focusing more on the workload characterization and the auto-tuning, and show our results on acoustic ray tracing for fly-over noise. We conclude that OpenCL's functional and (partial) performance portability are both instrumental for such an integrated solution.

3.18 Accelerating Algorithms for Biological Models and Probabilistic Model Checking

Anton Wijs (TU Eindhoven, NL)

License Creative Commons BY 3.0 Unported license
© Anton Wijs
Joint work of Bošnački, Dragan; Edelkamp, Stefan; Hilbers, Peter; Ligtenberg, Willem; Odenbrett, Max; Sulewski, Damian; Wijs, Anton

In this talk, I will discuss two applications using GPU computation capabilities from different domains. In the first application, biological networks resulting from genetic perturbation experiments are reduced in order to extract the vital information. We show that this boils down to computing the transitive reduction of weighted graphs. This can be performed ideally using GPUs. We explain how an extended version of the Floyd-Warshall algorithm can be parallelised effectively for the GPU. Through experimentation, we recorded speedups up to 92 times compared to a sequential implementation.

In the second application, we capitalize on the fact that most of the probabilistic model checking operations involve matrix-vector multiplication and solving systems of linear equations. Since linear algebraic operations can be implemented very efficiently on GPGPUs,

the new parallel algorithms show considerable runtime improvements compared to their counterparts on standard architectures. We implemented our parallel algorithms on top of the probabilistic model checker PRISM. The prototype implementation was evaluated on several case studies in which we observed significant speedup over the standard CPU implementation of the tool.

References

1 D. Bošnački, S. Edelkamp, D. Sulewski and A.J. Wijs, *Parallel Probabilistic Model Checking on General Purpose Graphics Processors*, International Journal on Software Tools for Technology Transfer 13(1):21–35, (2011).
2 D. Bošnački, M.R. Odenbrett, A.J. Wijs, W.P.A. Ligtenberg and P.A.J. Hilbers, *Efficient Reconstruction of Biological Networks via Transitive Reduction on General Purpose Graphics Processors*, BMC Bioinformatics 13:281 (2012).

4 Discussion Sessions

4.1 Domain specific languages for accelerators

Albert Cohen (ENS – Paris, FR)

License Creative Commons BY 3.0 Unported license
© Albert Cohen

The discussion covered three main questions:

1. Understanding the rationale for using or designing DSLs in the context of hardware accelerators.
2. Listing the most successful frameworks and illustrations of the approach, sharing and collecting people's experience.
3. Digging into the design and implementation of DSLs, including language embedding, staging, debugging.

The participants came from different horizons and included end-users (applications), language and compiler designers, and hardware accelerator experts. The field, the motivations, and experiences appeared to be very diverse. But some fundamentals and general lessons could be identified, aiming to maximize the usability of hardware accelerators, and to simplify the design of programming and development flows.

4.2 Verification techniques for GPU-accelerated software

Alastair F. Donaldson (Imperial College London, GB)

License Creative Commons BY 3.0 Unported license
© Alastair F. Donaldson

The aim of this session, involving roughly half the seminar participants, was to brainstorm the key issues surrounding verification techniques for GPU-accelerated software, to gather opinions about: the most important kinds of bugs that occur when programming GPUs, how verification technology might help in *optimization* of GPU kernels, and what infrastructure is most appropriate to use in building analysis tools. In additional we discussed a wish list for future developments in this area.

The following is a rough summary of the discussion.

Bugs affecting GPU programmers

- **Data races:** There was consensus that this is one of the most important classes of defect to be checked, with anecdotal evidence that engineers working for major GPU vendors also share this view. We discussed the problem of distinguishing erroneous data races from benign data races, and the idea of providing annotation facilities so that programmers could mark certain accesses as participating in benign races, to suppress false positives.
- **Bugs caused by compiler behaviour related to data races:** The discussion of benign races led to comments that even if the programmer intends a race to occur, this may lead to unpredictable behaviour from the compiler, since the semantics for racy code is in general not defined.
- **Barrier divergence:** It was suggested that this is not such a serious class of defect because it is usually fairly obvious when a barrier has been accidentally placed in non-uniform conditional code, but that nevertheless it is valuable to have techniques to check for this issue as barrier divergence can lead to deadlocks.
- **Bugs inherited from sequential programming:** Out-of-bounds accesses, accessing uninitialised memory, problems related to pointer type casting, etc., are just as problematic in GPU kernels as in sequential programs.
- **Floating point issues:** We discussed a bit the problems that can arise when discrete decisions are based on floating point computations; how this can lead to subtle problems and may be affected due to re-association of operations as a result of thread nondeterminism. We also debated the extent to which there is standardised floating point precision across architectures, and noted the inclusion of implementation-defined 'fast' transcendental functions in OpenCL.

Using verification techniques to help with code optimisation

The idea of using verification and symbolic execution to check for issues like memory bank conflicts in CUDA kernels has already been explored through the PUG and GKLEE tools at the University of Utah. We discussed the possibility for using verification to identify redundant synchronisation caused by cautious programming; i.e., figuring out that a particular barrier can be removed without inducing data races.

Infrastructure for building analysis tools

We discussed the advantages and disadvantages of analysis techniques that work at low-level program representations. Working at a 'close-to-the-metal' representation level for architecture X may help to find subtle bugs related to X, including bugs introduced by compilers, but may not be meaningful for other architectures. Working at a level such as LLVM IR allows the reuse of existing infrastructure (in this case CLANG/LLVM), but means that analysis is being performed with respect to one particular compiler framework, whereas a kernel may be compiled using a different framework. Source code is a common ancestor for all architectures, but does not take into account compiler effects at all; additionally source level analysis carries significant front-end implementation overhead which is avoided when working at the level of IR. A pragmatic compromise is to perform analysis on the "lowest-level common ancestor" representation. It was suggested that SPIR might be a suitable candidate.

The Ocelot framework[1] was also discussed as a promising base on which to build analysis tools.

Areas for future investigation

The following is a list of ideas and questions related to future work that were raised during the discussion.

- It is likely that future architectures and programming models will allow an increased level of dynamic memory allocation, meaning that techniques for heap analysis, such as separation logic, might play a larger role in GPU kernel verification.
- Relatedly, we discussed the role of linked data structures in kernel programming, including tricks programmers use to copy linked data structures into GPU global memory without re-wiring pointers; this seems like a challenging target for static analysis.
- Better techniques for analysis of floating point properties were mentioned several times; there is also wide demand for this in sequential program analysis.
- Can verification techniques be used for program repair, e.g., to suggest where to place barriers to eliminate data races, or to suggest where to move barriers to avoid barrier divergence?
- Most techniques so far have focussed on lightweight properties such as race-freedom. It would be interesting to look at functional verification, either of kernels themselves, or of applications that use kernels. In the latter case, modular verification would require specifications to be written describing the effects of a kernel.
- Race analysis of data-dependent kernels (to reason about, e.g., accesses of the form $A[B[e]]$ where A and B are shared memory arrays) may require some level of functional verification, even if this is not the ultimate goal.
- It could be interesting and useful to provide a mechanism for programmers to annotate barriers with light-weight assertions, to specify the intended use of a barrier.
- Can static analysis techniques help in understanding performance issues of GPU kernels, such as the possible gains from offloading, or the worst case execution time of a kernel?
- Techniques for cross-checking GPU and CPU implementations of an algorithm could be very useful; this has been explored to some extent through the KLEE-CL project.
- Can abduction be used to infer preconditions for kernels that are sufficient for ensuring race-freedom (or other properties)?

[1] https://code.google.com/p/gpuocelot/

Participants

- Jade Alglave
University College London, GB
- Adam Betts
Imperial College London, GB
- Albert Cohen
ENS – Paris, FR
- Christian Dehnert
RWTH Aachen, DE
- Dino Distefano
Queen Mary University of London, GB
- Alastair F. Donaldson
Imperial College London, GB
- Jeremy Dubreil
Monoidics Ltd. – London, GB
- Benoit Dupont de Dinechin
Kalray – Orsay, FR
- Ganesh L. Gopalakrishnan
University of Utah, US
- Sebastian Hack
Universität des Saarlandes, DE
- Lee Howes
AMD – Sunnyvale, US
- Marieke Huisman
University of Twente, NL
- Christina Jansen
RWTH Aachen, DE
- Joost-Pieter Katoen
RWTH Aachen, DE
- Jeroen Ketema
Imperial College London, GB
- Alexander Knapp
Universität Augsburg, DE
- Georgia Kouveli
ARM Ltd. – Cambridge, GB
- Alexey Kravets
ARM Ltd. – Cambridge, GB
- Anton Lokhmotov
ARM Ltd. – Cambridge, GB
- Roland Meyer
TU Kaiserslautern, DE
- Cedric Nugteren
TU Eindhoven, NL
- Zvonimir Rakamaric
University of Utah, US
- Oliver Reiche
Univ. Erlangen-Nürnberg, DE
- Philipp Rümmer
Uppsala University, SE
- Ana Lucia Varbanescu
TU Delft, NL
- Sven Verdoolaege
INRIA, FR
- Jules Villard
University College London, GB
- Heike Wehrheim
Universität Paderborn, DE
- Anton Wijs
TU Eindhoven, NL
- Marina Zaharieva-Stojanovski
University of Twente, NL
- Dong Ping Zhang
AMD – Sunnyvale, US