Improving Accuracy of Source Level Timing Simulation for GPUs using a Probabilistic Resource Model

Christoph Gerum, Wolfgang Rosenstiel and Oliver Bringmann
University of Tübingen, Sand 13, 72076 Tübingen
{*gerum,rosen,bringman*}@*informatik.uni-tuebingen.de*

***Abstract*—After their success in the high performance and desktop market, Graphic Processing Units (*GPUs*), that can be used for general purpose computing are introduced for embedded systems on a chip (*SOCs*). Due to some advanced architectural features, like massive simultaneous multithreading, static performance analysis and high-level timing simulation are difficult to apply to code running on these systems. This paper extends a method for performance simulation of GPUs. The method uses automated performance annotations in the application's OpenCL C source code, and an extended performance model for derivation of a kernels runtime from metrics produced by the execution of annotated kernels. The final results are then generated using a probabilistic resource conflict model. The model reaches an accuracy of 90% on most test cases and delivers a higher average accuracy than previous methods.**

I. Introduction

Modern embedded applications need to solve very complex computations in a limited amount of time. This includes for example advanced driver assistance systems like lane detection and obstacle avoidance, augmented reality or multimedia applications. These ever increasing performance demands have led to the introduction of a number of embedded systems on a chip (*SoC*), which not only use powerful multicore systems but also contain powerful GPUs that can be used as hardware accelerators for compute intensive tasks. Examples of such systems include the NVIDIA Tegra K1 [1] or systems containing ARM MALI GPUs [2].

In the development phase of these systems, no hardware implementation is available. Therefore developers often use virtual prototypes for early validation of the timing behaviour, hardware software codesign and design space exploration of hardware/software systems. Much effort has been invested to make these virtual prototypes for single and multiprocessor systems faster, more accurate and easier to develop. The emerging technology of heterogeneous systems including GPU cores is mostly not considered by current academic and commercial virtual platforms.

One performance simulation technique that has gained interest in the recent years is source level timing simulation. Source level timing simulation works by instrumenting each basic block of an applications source code with function calls to simulate the timing behaviour of the corresponding part of the applications binary code on a specific target system. Source level timing simulation is one of the fastest simulation techniques that are currently available. This high simulation speed is reached because the granularity of the performance simulation is lifted from instruction level to basic block level. And the annotated source code can be optimized more easily for the architecture of the simulation host compared to binary translation of the applications compiler generated binary code.

So far there is no performance simulation technique available that allows a simulation of systems containing GPU cores at a comparable level of abstraction and with a comparable simulation speed and accuracy. The main contributions of this work are:

1) A detailed simulation model for resource conflicts that better approximates the runtime of GPU accelerated tasks than previous work [3].
2) This work especially focuses on the timing effects of limited access bandwidth to the register file.
3) An extended evaluation which shows the benefits and limits of this simulation technique.

The remainder of this paper is structured as follows. Section II describes the state of the art considering performance modelling of GPUs. Section III gives a short introduction to the microarchitecture of current GPUs and motivates our performance modeling. Section IV and its subsections provide a detailed overview of the methods used for performance analysis and simulation and gives an in depth explanation of our stochastic model for resource conflicts. In Section V we give some insight in the implementation details of our simulation method. Section VI gives a speed and accuracy comparison of our method with a state of the art cycle accurate performance simulator.

II. Related Work

The only previous work that does source level performance simulations for GPU cores is [3]. Our work builds on this system but uses statistical modelling of resource contention for some of the resources of the pipeline. This in many cases improves the accuracy of the performance simulations. Other current examples for source level performance simulation are [4], [5], [6], [7], [8] and [9]. They mostly have been developed for the simulation of single core systems. Some of them have been used to simulate multicore systems with a limited number of cores. None of them is directly applicable to simulate the complex architectural features of current GPUs. And none of them uses parallelism available on the simulation host to speed up the simulations.

978-1-4673-7311-1/15/$31.00 ©2015 IEEE

Considering other techniques for performance simulation and analysis of GPU cores there are some techniques available. The most widely used tool for performance simulations of GPUs is *gpgpu-sim* [10] which uses a slow interpretive simulator for functional simulation of GPUs coupled with a detailed micro-architectural simulator. This leads to slow simulations, which often make simulation of real world applications infeasible. Other techniques for performance estimations of GPU cores were presented in [11] and [12]. Both of them extract performance relevant metrics like instruction counts, instruction traces or accessed memory addresses during the simulation of a program. These metrics are then analyzed by a performance model to get an estimate on the execution time of an application. Huang et. al. [13] use an interval analysis on traces generated from a functional simulator to get an analytical estimation for software execution time on GPUs. Because these methods still rely on a functional simulation of the application on an instruction set simulator their simulation performance is still relatively low. The authors in [14] use a cycle accurate simulator to get basic block execution times and basic block execution traces. These traces are then combined to calculate a measurement based worst case execution time. This approach is very interesting as it allows an approximation of the global worst case timing. However the use of multiple instruction set simulations with varying input parameters makes this approach infeasible on realistic problem sizes. In [15] a purely analytical performance model is proposed, but the experimental evaluation is very limited and many metrics used to derive an execution time estimate are currently not obtainable using source level simulations.

III. GPU-Microarchitecture and Execution Model

Programming models for GPUs require a manual partitioning of code that is executed on the traditional CPUs of a system (the *host*) and code that is executed on the *device*. A device may be the same CPU core as the host but can also be a GPU or other accelerators. Code that is intended to run on the host mostly stays in its current form, while code that should run on a device is mostly rewritten in a specialized programming language that allows a higher degree of parallelism than traditional C/C++. The most common languages for this purpose, CUDA C/C++ and OpenCL C, are both based on ANSI C/C++. As OpenCL is available on many platforms, and CUDA is quite closely locked to NVidia GPUs, our framework is based on the OpenCL programming environment. Functionality to be run on a device is formulated as a number of massively data-parallel functions called *kernels*. Each kernel consists of 10s to 1000s of *threads* or *work items*. The exact number of threads in most cases depends on the sizes of the input and output. All threads of a kernel form the *global work group*. The programmer can choose subsets of kernels which are called *local work groups*.

Current GPUs execute all threads of a kernel using several layers of parallelism. The outermost layer consists of a number of streaming multiprocessors. When there are multiple streaming multiprocessors, this layer closely resembles traditional multiprocessing. The threads scheduled to different multiprocessors need to belong to different local work groups. The remaining levels of parallelism are handled within the pipeline of one streaming multiprocessor. The levels of parallelism are

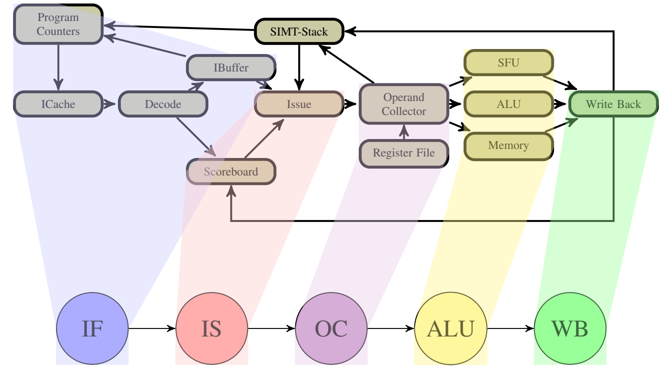

Fig. 1: GPU-microarchitecture and our pipeline model

simultaneous multithreading and *warp level parallelism*. *Warps* consist of the instructions of multiple threads from the same local work group. While the warp size may vary depending on the size of local work groups, the preferred and maximum warp size on Nvidia GPUs has been 32 threads for several generations. The threads of a warp always execute the same instruction in lockstep but are allowed to branch independently. As the instructions are allowed to branch independently, a so called *branch divergence* can occur. Branch divergence is handled in hardware by executing each path sequentially and masking those operations which did not take the currently selected path. As warps execute the instructions of several threads in lockstep, we use the term *warp instruction* to mean the instruction that is currently executed by all threads in the warp. Each streaming multiprocessor handles multiple warps concurrently using fine-grained multithreading. All warps of a local work group need to run on the same streaming multiprocessor, but depending on the resource usage of a local work group, multiple local work groups might be handled by the same streaming multiprocessor concurrently.

Warp instructions are handled by a pipeline structured as the one in Figure 1. The frontend fetches two instructions of one warp at a time and places the decoded instructions in an instruction buffer. If more than 2 warps have ready instructions, instructions are scheduled in a round robin manner. The issue stage of the pipeline issues instructions to the operand collector stage of the pipeline. This stage reads the input operands from a banked register file. The warp instructions are executed on different functional units, depending on the instruction type. In our model there are two functional units to handle arithmetic and logic instructions (ALU), one special functional unit (SFU) to handle specialized instructions like trigonometric operations and one memory unit to handle load store instructions. After execution, the results are written back to the register file and instructions waiting for the results are notified through the scoreboard.

In this work we especially focus on modelling the performance impact of accesses to the register file in the operand collector and writeback stage of the pipeline. For access to the register file GPUs use a operand collector structure described in [16]. This part of the pipeline is shown in more detail in Figure 2.

Upon issue Instructions are first stored in operand collector units. In the following cycle instructions begin to copy their

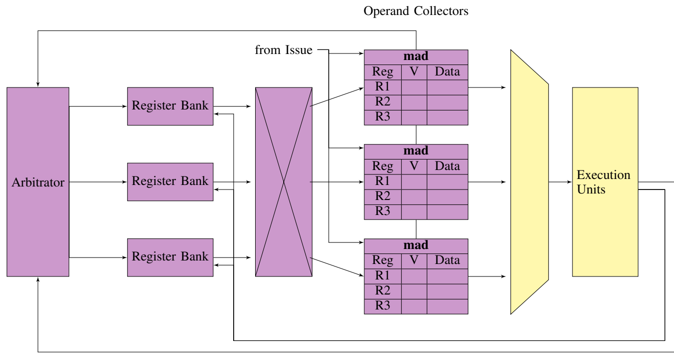

Fig. 2: Implementation of Register Accesses in GPU-Architectures

operands from a banked register file to the collector units. Instructions read at most one operand per cycle, meaning the collector units are occupied by the instruction for at least 1 cycle + number of operands. An arbitrator ensures that each register bank can only be accessed by one collector unit at a time. After all operands of an instruction have been read the instruction is dispatched to the execution units of the pipeline. After execution the results are written back to the register file. Writebacks can conflict with other writebacks as well as with reads from the Operand Collectors. If conflicts occur the arbitrator schedules the conflicting accesses in a fair round-robin order.

IV. Source-Level Performance Simulation for GPUs

Source level performance simulation of GPU cores allows performance estimation for GPU kernels without the need of a slow functional simulation of the GPUs instruction set architecture. The structure of our simulation framework is shown in Figure 3. The source code is first translated by an OpenCL compiler to PTX assembly code for an NVidia GPU architecture. As the PTX assembly needs to contain debugging information for the following matching steps to work, we cannot use the official Compiler from NVidia but use a compiler toolchain based on clang and LLVM [17]. We then construct a control flow graph from the source code as well as from the assembly code. Both control flow graphs are used to match the corresponding basic blocks on the source and binary level. A detailed description of the matching step is out of scope for this paper. The algorithm used for matching is similar to the one in [6]. The CFG on the binary level is also used to do a low-level optimistic pipeline analysis. This analysis extracts latencies for the execution of each binary level basic block. The results of the binary to source matching and the low-level pipeline analysis are used to create a version of the original source code with timing and resource usage annotations. These annotations enable a fast “simulation” of the pipeline behavior through execution on any OpenCL compatible device. Through the execution on any OpenCL device performance simulations can be carried out without availability of a specific GPU, provided that a performance model for the simulated GPU is available. The timing behavior simulated by the native execution on a device is not considering effects of resource sharing due to the simultaneous multithreading on a GPU

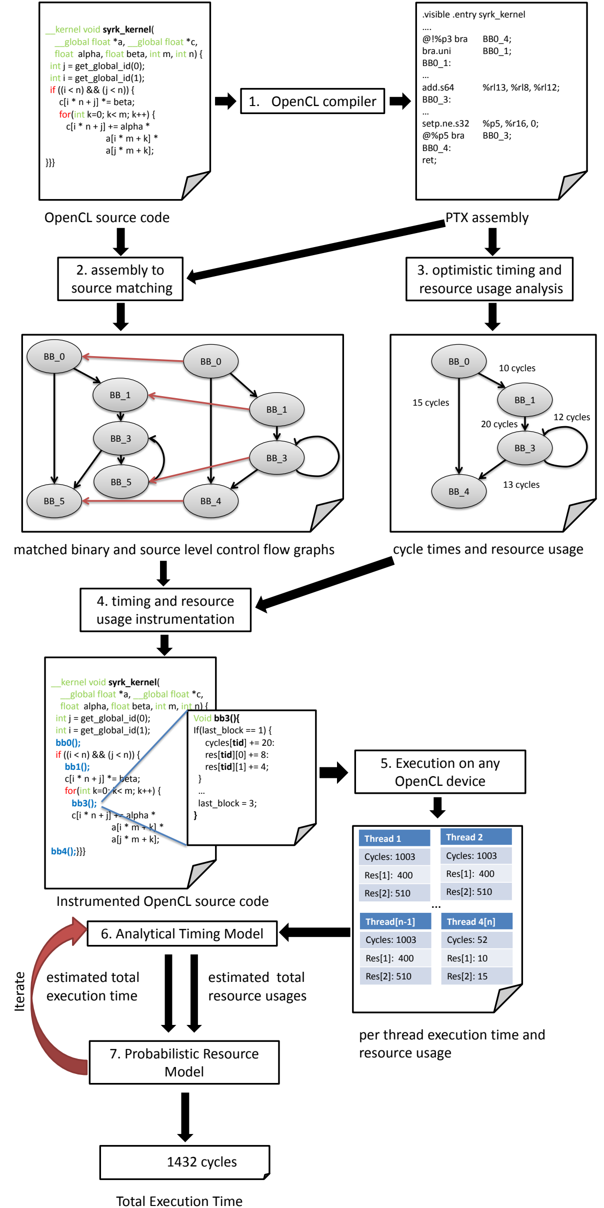

Fig. 3: Structure of the proposed simulation framework

core. These effects are incorporated into our model by the step called *Analytical Timing Model* in Figure 3. In Section IV-A we describe the performance model to simulate the timing behaviour of GPU-Architectures. It does not use probability theory to approximate the execution time of applications and instead approximates a lower bound for the execution time by taking best case assumptions. The results of the Analytical Timing Model are then used to approximate the number of conflicts for some resources with a *Probabilistic Resource Model*. This model refines the resource usage information from the analytical model by taking conflicting accesses to shared

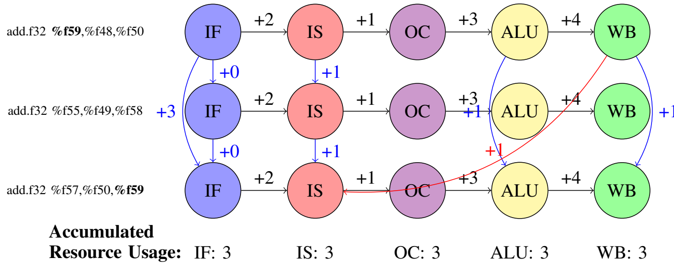

Fig. 4: Example of our pipeline analysis

resources into account. The refined resource usages are then used to reiterate the analytical model until resource usages no longer change. The estimated execution time of the last iteration is the final estimate of the execution time.

A. The Optimistic Analytical Performance Model

During the static analysis phase of the framework we build pipeline execution graphs for each basic block in the system. The timing is then determined by searching for the longest path in the pipeline execution graph. Additional to the determination of the execution time of basic blocks we analyze the resource usage time of each basic block. The resource usage time is the time one resource of the pipeline is occupied by an instruction and cannot be used by an other instruction. In Figure 4 we show the pipeline execution graphs and the analyzed resource usages. The static execution time as well as the analyzed static resource usages for each basic block are annotated back to the original source code. Execution of the annotated source code on any OpenCL capable compute device gives the accumulated resource usages. The accumulated per thread resource usages are then reduced to the accumulated resource usages of the whole kernel.

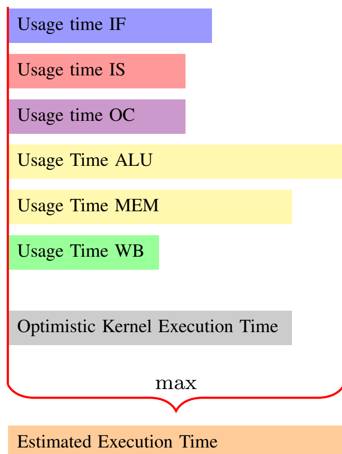

Fig. 5: Illustration of the optimistic analytical model

In Figure 5 we illustrate the result of the reduction step of the analytical model. In the optimistic Analytical model we assume perfect scheduling for all resources. That means conflicting accesses to resources only happen if the resource is fully saturated. In this case the estimate for the final execution time is solely the maximum over the optimistic total execution time and the resource usages. For details of the analytical model we refer to [3].

B. Probabilistic Extension of the Analytical Timing Model

Initial experiments with the cycle accurate GPU simulator `gpgpu-sim` showed that the number of banks in the register file and the number of collector units influence the execution time of GPU-Kernels. But there is no direct way of determining the number of bank conflicts from the metrics used by the optimistic execution model. We therefore choose to model the performance impact of the register accesses using a probabilistic model.

Let X_i be a family of random variables that give the bank number for each memory access i. We assume that register bank accesses are uniformly and independently distributed among the memory banks. This assumption does not hold if we consider only a single warp. Register accesses from a single warp instruction do never conflict with each other.

But since the actual bank numbers used for a register name are determined by a hash function from the register name and the warp id at runtime and the pipeline is used by multiple warps concurrently, uniform and independent distribution can be used at least as a rough approximation of the actual distribution. Under these assumptions the register bank conflict model should be more accurate if there are more warps concurrently on the pipeline.

If there are b register banks in the system this leads us to the probability of an access to bank k of:

$$P[X_i = k] = \frac{1}{b}$$

If we have m simultaneous accesses the probability that exactly n accesses go to bank k is:

$$P[Y_k = n] = \binom{m}{n} \left(\frac{1}{b}\right)^m \left(1 - \frac{1}{b}\right)^{m-n}$$

The expected number of simultaneous accesses is then by definition of the expected value:

$$E[Y_k] = \sum_{i>0}^{m} i \cdot P[Y_k = i]$$

The number of conflicts is one less than the number of accesses to the register bank. So the estimated number of conflicts per bank is:

$$n_{\text{conflicts}_k} = E[Y_k] - 1$$

The total number of conflicts for all register banks in the system is then the sum over all banks in the system.

$$n_{\text{conflicts}} = \sum_{k=0}^{b} n_{\text{conflicts}_k}$$

And as the expected number of conflicts is the same for all banks in the system this can be simplified to:

$$n_{\text{conflicts}} = b \cdot n_{\text{conflicts}_0}$$

If this model should be applied to the GPU pipeline there are still two important points missing. First the number of simultaneous accesses m is not known. The second missing point is the execution time estimation from per clock register bank conflicts. Our solution for this problem is the following. We first determine the number of read and written registers per basic block during static analysis. This information is then annotated back in the original source code in addition to the metrics described in section IV-A. Execution of the annotated source code, accumulates the total number of register read and write accesses. The calculation of the total number of register read and write accesses is implemented as the resource usage calculation. Registers are always shared between all threads in a warp. This means the total number of read or written registers of a warp is approximately the maximum number of read or written registers of all threads in the warp. Instructions from different warps do never share registers. Therefore the total register usage is the sum over the register usages of all warps.

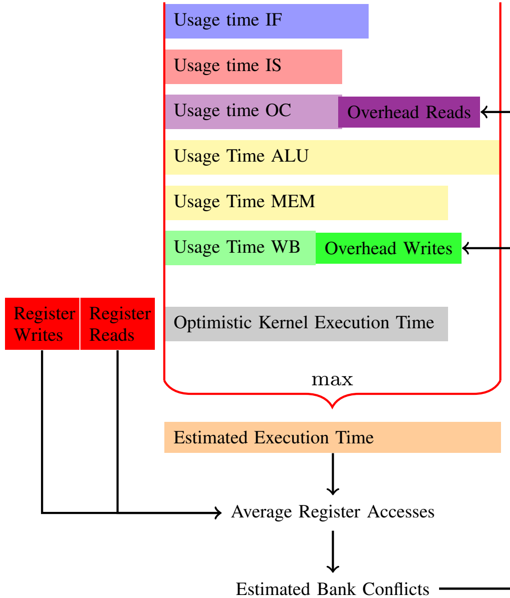

Fig. 6: Illustration of the new resource conflict model

The final steps of the model are shown in Figure 6. As a first step the optimistic execution time is calculated. We then use this time to approximate the average number of accesses to the register file by dividing the total number of accesses through the number kernel execution times and rounding the result down to the next integer. This result and the number of register banks is then used together with the probabilistic model to estimate the average number of register bank conflicts per cycle. An estimate of the total number of conflicts is then calculated using the number of conflicts per cycle and the optimistic estimate of the number of cycles. Multiplying this result with the number of register reads gives us the additional usage time of the operand collectors. The additional usage of the writeback stage of the pipeline is calculated by multiplying it with the number of register writes. We then update the usage information of the Operand Collectors and the writeback stage and calculate the new estimate of the execution time using the updated usage information. The algorithm is then repeated with the new execution time until a fix point is reached or up to a user defined maximum number of iterations.

V. Implementation Details

The performance simulation framework has been implemented as a shared library implementing the part of the OpenCL runtime API that is responsible for compiling and running OpenCL code on a GPU. The idea of implementing the simulation framework this way has been taken from `gpgpu-sim`. It allows to seamlessly replace our simulator with `gpgpu-sim` or the host OpenCL implementation just by changing the value of the Linux `LD_PRELOAD` environment variable. The functions implemented in this library are `clCreateProgramWithSource`, `clBuildProgram` and `clEnqueueNDRangeKernel`.

`clCreateProgramWithSource` just stores the OpenCL source code it receives as a parameter for further analysis and annotation purposes on the hard disc of the simulation host.

`clBuildProgram` first calls an OpenCL C compiler for the target architecture to get binary code for the target GPU. It then calls our annotation and static analysis framework with the original version of the source code and the target binary to produce an instrumented version of the source code. The implementation then uses `clCreateProgramWithSource` and `clBuildProgram` of the simulation host to produce an executable version of the instrumented kernel.

Our version of `clEnqueueNDRangeKernel` first allocates buffers for the additional performance metrics. It then starts the instrumented kernel with its original parameters and the additional performance metrics. After the execution of the instrumented Kernel, the result buffers are transferred to the simulation host and evaluated using the analytical performance models. Finally the memory for the performance metrics is freed and the execution of the program is resumed.

VI. Results

The described extensions have been integrated in the model of [3]. For the Evaluation of our performance model we used benchmarks from then suites Rodina [18] and polybench-gpu [19]. The benchmarks from polybench-gpu are naive implementations of simple mathematical kernels mostly functions from the BLAS linear algebra library. The applications from Rodinia on the other hand have a much higher complexity. These benchmarks use full applications with kernels optimized for high execution speed on GPU devices. Table I gives a short description of the benchmarks used for evaluation. We additionally list the classification of the benchmark in one of the 13 Berkley dwarfs [20]. All benchmarks have been compiled to *ptx*-Assembly using *clang* and *llvm* [17]. With the compiler flags `-O3 -g` to enable most compiler optimizations and to generate debug information for the binary to source matching algorithm.

TABLE I: Benchmarks used for evaluation of the simulator

Benchmark	Description	Dwarf	Suite
backprop	Training of a multilayer perceptron	Unstructured Grid	rodinia
bfs	Parallelized breadth first search in a Graph	Graph Traversal	rodinia
bplustree	Parallelized lookup in a B+-Tree	Graph Traversal	rodinia
gaussian	Gaussian elimination on a 4x4-Matrix	Dense Linear Algebra	rodinia
kmeans	K-Means clustering of a large dataset	Dense Linear Algebra	rodinia
lud	LU decomposition	Dense Linear Algebra	rodinia
nn	Search of nearest neighbour in a database with hurricane data	Dense Linear Algebra	rodinia
nw	Needlman Wunsch algorithm for optimization of DNA sequence alignment	Dynamic Programming	rodinia
particle	Object tracking using a particle filter	Dense Linear Algebra	rodinia
streamcluster	Clustering of a large Dataset	Dense Linear Algebra	rodinia
2DConv	2DConvolution using a 3x3 kernel	Structured Grid	polybench-gpu
2MM	Multiplication of 2 Matrices	Dense Linear Algebra	polybench-gpu
3DConv	3Dconvolution using a 3x3x3 Kernel	Structured Grid	polybench-gpu
3MM	Multiplication of 3 Matrices	Dense Linear Algebra	polybench-gpu
FDTD2D	Solving of two dimensional differential equations using finite-difference time-domain method	Dense Linear Algebra	polybench-gpu
GEMM	Generalized matrix multiplication $C \leftarrow \alpha AB + \beta C$	Dense Linear Algebra	polybench-gpu

The reference for the simulations is the cycle accurate simulator `gpgpu-sim` configured to simulate a single core of a NVIDIA Fermi architecture GPU. Since our simulator currently does not contain a memory model we activated the perfect memory mode of the `gpgpu-sim`. In this mode memory accesses are always simulated as hits in the first level cache.

We first evaluate the accuracy of the estimated number of operand collector conflicts. For this purpose we compare the estimated number of register bank conflicts with the actual number of operand collector conflicts as simulated by gpgpusim. As the original version of gpu-sim does not report the number of operand collector conflicts we made a slight modification to the simulator's source code to include this number in its output. The comparison of the simulated number memory conflicst according to gpgpu-sim with the number of conflicts given by the performance estimator included in our estimation framework. The results of this comparison are shown in table II. The table also shows the total number of warps in each kernel and the number of warps that are executed in parallel.

TABLE II: Accuracy evaluation of the probabilistic register bank conflict estimation

Name	Warps	Parallel Warps	Estimated Conflicts *Our model*	Simulated Conflicts *gpgpusim*	Relation
backprop0	32768	48	556794	301251	1.8483
backprop1	32768	48	282836	221409	1.2774
bfs0	31256	48	83203	53033	1.5689
bfs1	31256	48	48776	21489	2.2698
bplustree0	48000	48	513902	500644	1.0265
bplustree1	80000	48	538226	323143	1.6656
gaussian0	1	1	85	0	nan
gaussian1	8	8	135	25	5.4000
kmeans0	494020	8	49833766	15642545	3.1858
kmeans1	494020	8	366446960	288321549	1.2710
lud0	1	1	10901	361	30.1967
lud1	63	8	59075	31149	1.8965
lud2	31752	8	1267467	856854	1.47921
nn0	1338	48	6733	6873	0.9796
nw0	1	1	2680	40	67.0000
nw1	2	2	2680	115	23.3043
streamcluster0	2048	48	2632	2061	1.2770
streamcluster1	98304	48	126336	95159	1.3276
2MM0	32	32	2059	1883	1.0935
3DConv0	32	32	590	790	0.7468
3MM0	8192	48	7597734	6991199	1.0868
GEMM0	8192	48	11358303	10911143	1.0410

The experimental data confirms the assumption from Section IV-B. If there is only one or a few warps executed on the pipeline in parallel, the model severely overestimates the number of memory bank conflicts. On the other hand in all of these cases the absolute number of register bank conflicts is relatively small and the number of register bank conflicts does not influence the number of estimated cycles for all benchmarks where this overestimation happens. For the other benchmarks the number of register bank conflicts is approximated comparatively well. The mean of the relative errors for benchmarks where there is an influence of the estimated cycles is 1.3 this still indicates a slight overstimation of the register bank conflicts by our model, but the actual number of register bank conflicts is still aproximated quite well. This small average overestimation of register bank conflicts can partially be explained by the interaction with the optimistic performance estimation model. If the final execution time is still below the execution time on the reference hardware, the number of register bank conflicts will be overestimated.

In Figure 7 we show the accuracy results considering the simulated instructions. As GPUs support predicated execution the blue bar shows the portion of instructions that actually gets executed. The red part of the bar shows the portion of instructions that are fetched but not executed based on the predicate registers of the GPU according to our performance simulator. The reference is the sum of executed and simulated instructions according to gpgpusim. The results diverge by less than 10% for all kernels except for the second kernel of the benchmark `lud`. A closer examination of the results reveals that the number of executed instructions is 98% percent accurate. This shows that the source to binary matching for this kernel is as accurate as for the other kernels. The overhead of almost 50% is attributed almost completely to our simplified handling of divergent branches. This problem can be handled by a simulation approach taking divergent branches into account, but this is out of the scope of this paper.

In Figure 8 we show an accuracy comparison of the purely optimistic baseline model and the probabilistic model described in this paper. The optimistic baseline is shown in blue while the improvements of the probalistic extension are shown in red. As expected the purely optimistic model always

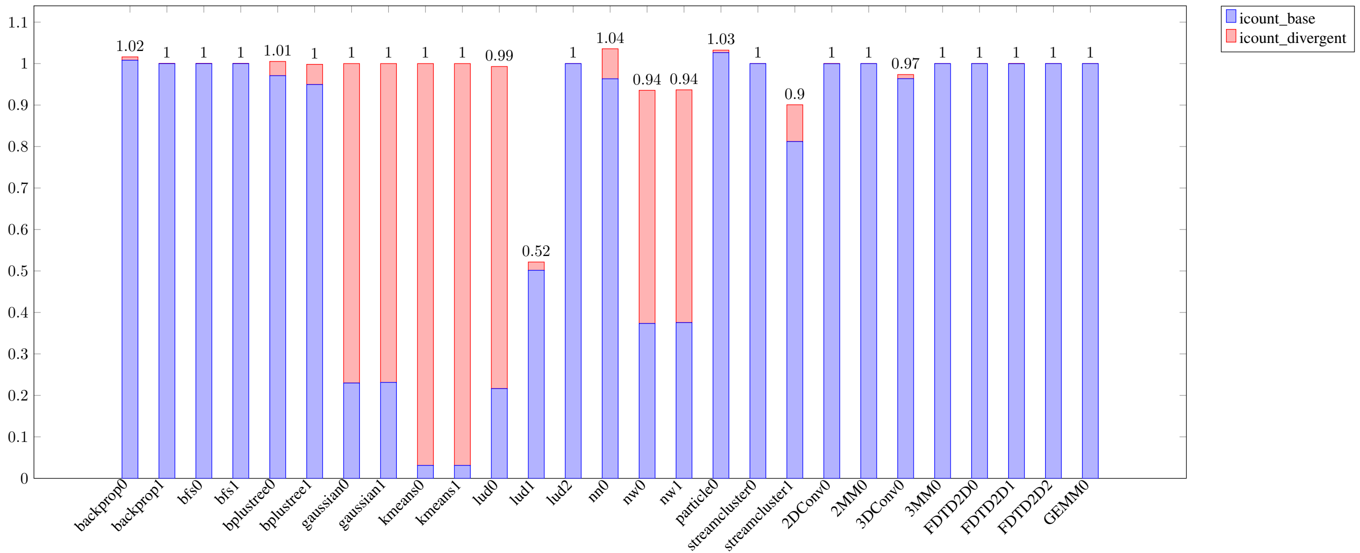

Fig. 7: Simulated instructions normalized by the instruction count from gpgpu-sim

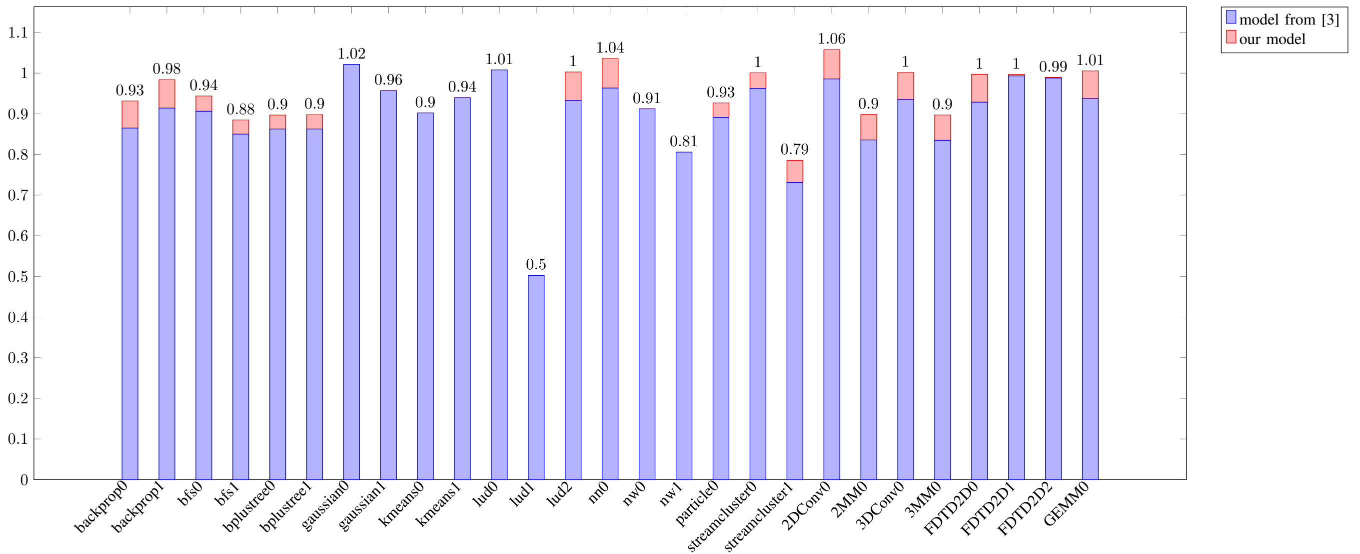

Fig. 8: Simulated cycles normalized by the cycle count from gpgpu-sim

produces lower or the same execution times as the probabilistic extension. The accuracy is improved for all but one benchmarks for which the probabilistic model shows an effect. The one exception, where the probabilistic Model reduces accuracy is 2DConv0. In this case the accuracy changes from 0.98 to 1.06 which is a reduction of the simulation accuracy by 4%. The average simulation accuracy is 94%. All but one of the benchmarks with nearly 100% accuracy in the instruction count comparison show a cycle count accuracy between 90% and 110%. The results are still optimistic, as accesses to other resources are still assumed to be scheduled perfectly and the impact of different scheduling policies is not considered.

The speedups compared to a cycle accurate simulation using *gpgpu-sim* range between 331 for the kernel 3DConv0 and 37348 for the kernel kmeans1. Which is in the same order of magnitude as in [3] which means the simulation speed has not been slowed down significantly by the additional model. For the speed comparison both simulators run on an Intel(R) Core(TM) i7-4770K CPU.

VII. Conclusion and Future Work

In this paper we have presented a method to increase the accuracy of source level simulations for GPU-cores with a probabilistic model of conflicting accesses to the banked register file. The experimental results show that the method improves the accuracy by reducing the time difference compared to a cycle accurate simulator below 10% for most tested Benchmarks. We assume that a similar approach can be used to model the timing of many modern computer architectures.

In our future work we will address the remaining problems of this simulation approach by integrating simulation

methodologies for the simulation of diverging branches and the memory subsystem. Our next step will be the integration of a performance model for the memory subsystem. We think that similar methods than the ones presented in this paper can be used to model the timing behaviour of banked resources in the memory subsystem e.g. the L2-cache, the local memory and the global DRAM, while the simulation of caches needs the integration of a Cache simulator. The integration of a memory model will also allow us to compare our model with measurements taken on a real GPU. We would also like to integrate the simulator in a highlevel virtual platform solution like [7] to enable full system simulations of embedded systems containing GPUs.

VIII. Acknowledgments

This work was partially funded by the State of Baden-Württemberg, Germany, Ministry of Science, Research and Arts within the scope of a Cooperative Research Training Group (EAES).

References

[1] Nvidia, “NVIDIA Tegra K1 A New Era in Mobile Computing,” pp. 1–26.

[2] R. Mijat, “Take GPU Processing Power Beyond Graphics with Mali GPU Computing.” 2012.

[3] C. Gerum, O. Bringmann, and W. Rosenstiel, “Source Level Performance Simulation of GPU Cores,” in *Design Automation and Test Europe*, (Grenoble), 2015.

[4] K. Lu, D. Muller-Gritschneder, and U. Schlichtmann, “Hierarchical control flow matching for source-level simulation of embedded software,” *2012 International Symposium on System on Chip (SoC)*, pp. 1–5, Oct. 2012.

[5] S. Stattelmann, O. Bringmann, and W. Rosenstiel, “Fast and accurate source-level simulation of software timing considering complex code optimizations,” *Proceedings of the 48th Design Automation Conference on - DAC ’11*, p. 486, 2011.

[6] S. Stattelmann, O. Bringmann, and W. Rosenstiel, “Dominator homomorphism based code matching for source-level simulation of embedded software,” in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS ’11*, (New York, New York, USA), p. 305, ACM Press, 2011.

[7] H. Posadas, S. Real, and E. Villar, “M3-SCoPE: Performance Modeling of Multi-Processor Embedded Systems for Fast Design Space Exploration,” in *Multi-objective Design Space Exploration of Multiprocessor SoC Architectures: the MULTICUBE Approach* (C. Silvano, W. Fornaciari, and E. Villar, eds.), pp. 19–51, Springer, 2011.

[8] Z. Wang and A. Herkersdorf, “An efficient approach for system-level timing simulation of compiler-optimized embedded software,” in *Proceedings of the 46th Annual Design Automation Conference - DAC ’09*, (New York, New York, USA), p. 220, ACM Press, 2009.

[9] A. Gerstlauer, S. Chakravarty, M. Kathuria, and P. Razaghi, “Abstract System-Level Models for Early Performance and Power Exploration,” in *Asia and South Pacific Conference on Design Automation*, 2012.

[10] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, Apr. 2009.

[11] J. Lai and A. Seznec, “Break down GPU execution time with an analytical method,” *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation Methods and Tools - RAPIDO ’12*, pp. 33–39, 2012.

[12] A. K. Parakh, M. Balakrishnan, and K. Paul, “Performance Estimation of GPUs with Cache,” *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pp. 2384–2393, May 2012.

[13] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, “GPUMech: GPU Performance Modeling Technique Based on Interval Analysis,” in *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 268–279, 2014.

[14] A. Betts and A. Donaldson, “Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis,” *2013 25th Euromicro Conference on Real-Time Systems*, pp. 193–202, July 2013.

[15] L. Ma, R. D. Chamberlain, and K. Agrawal, “Performance Modeling for Highly-threaded Many-core G,” in *25th International Conference on Application-Specific Systems, Architectures and Processors*, 2014.

[16] S. Liu, E. Lindholm, M. Y. Siu, and S. Clara, “Operand Collector Architecture,” 2010.

[17] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, (Palo Alto, California), Mar. 2004.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, vol. 2009, pp. 44–54, Ieee, Oct. 2009.

[19] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, “Auto-tuning a high-level language targeted to GPU codes,” in *2012 Innovative Parallel Computing (InPar)*, pp. 1–10, Ieee, May 2012.

[20] K. Asanovic, B. C. Catanzaro, D. a. Patterson, and K. a. Yelick, “The Landscape of Parallel Computing Research : A View from Berkeley,” tech. rep., 2006.