Source Level Performance Simulation of GPU Cores

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

***Abstract*—Graphic processing units (*GPUs*) contain a lot of complex architectural features, which make performance analysis and simulation of applications using them for general purpose computation very difficult. Especially when trying to do performance simulations at a higher abstraction level than interpreted instruction set simulators these features are not handled accurately by state of the art simulation techniques. This paper proposes a method for source level performance simulation of the microarchitecture of a GPU core that provides high enough simulation speeds to make testing of large application scenarios possible.**

I. Introduction

The increased performance demands of many modern embedded applications have led to the development of a number of embedded platforms like the NVidia Tegra K1 that not only include powerful scalar or superscalar CPUs but also have Graphic Processing Units usable as accelerators for general purpose computations [1]. In embedded systems design, timing simulations are a very common tool for early verification of the timing behavior of an embedded system. The early availability of these performance simulations in the design process allows Hardware/Software Codesign without in silicon prototypes of the hardware components. One technique for performance simulations, that in recent years has gained popularity, is source level performance simulation. Source level performance simulation reaches high simulation performance and is available early in the design process. This paper proposes a method for source level performance simulation of a GPU core. To simulate the timing behavior, the execution time of parts of a program is first analyzed using a static analysis tool. This information is then annotated back in the original source code. Timing simulations are carried out by executing the annotated source code on any device and estimating the final execution time from the accumulated performance metrics obtained from the native execution of the annotated source code using an analytical model. The simulation technique is aimed at being fast enough to simulate long running application scenarios.

The rest 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. Section V gives a speed and accuracy comparison of our method with a state of the art performance simulator.

II. Related Work

Latest examples for source level performance simulations are presented in [2], [3], [4], [5] and [6]. All of them focus on source level simulation of classic scalar CPU cores. To the best of our knowledge there is no previous work considering source level timing simulation of GPU cores. The most widely used tool for performance simulations of GPUs is *gpgpu-sim* [7] which uses a slow interpretive simulator for functional simulation of GPUs coupled with a detailed microarchitectural 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 [8] and [9]. 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. Because these methods still rely on a functional simulation of the application on an instruction set simulator their simulation performance is still relatively low. One technique presented in [10] uses 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 but the use of multiple instruction set simulations with varying input parameters makes this approach infeasible on realistic problem sizes.

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, CUDA C/C++ and OpenCL C, for this purpose 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.

978-3-9815370-4-8/DATE15/©2015 EDAA

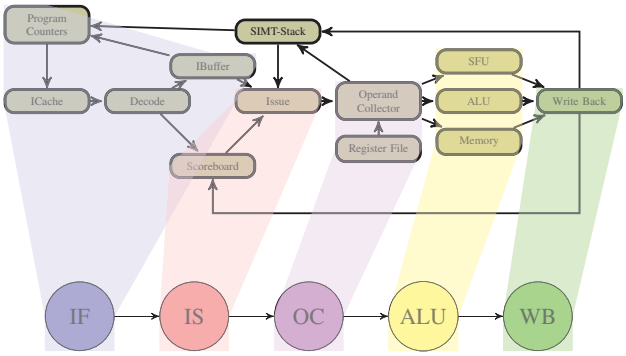


Fig. 1: GPU-microarchitecture and our pipeline model

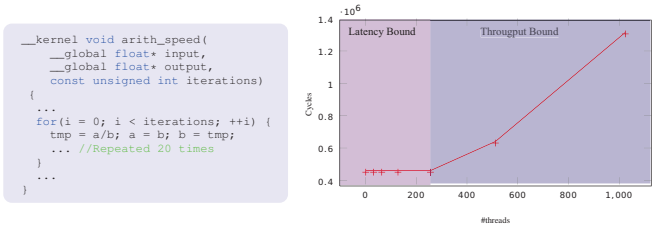


Fig. 2: Microbenchmark to demonstrate the timing behavior of a GPU-Microarchitecture

The remaining levels of parallelism are handled within the pipeline of one streaming multiprocessor. These are considered by our performance simulation. The levels of parallelism are *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. Figure 2 shows a simple microbenchmark and its execution time on a GeForce GTX480 architecture as simulated by the simulator `gpgpu-sim`. The kernel of the benchmark executes a loop containing 30 data dependent floating point divisions. The kernel is executed with a varying number of threads and its execution time on the simulator is measured. Up to 256 threads the execution time of the kernel remains almost unchanged (1 thread: 460218 cycles, 256 threads: 460263).Up to 32 threads this is explained by the warp level parallelism. The GPU executes a warp of 32 threads in lockstep so the execution time for up to 32 threads is the same as the execution time with one thread. Over 32 threads the GPU interleaves multiple warps using simultaneous multithreading. Due to the data dependencies between the divisions up to 256 threads e.g. 8 warps can be executed interleaved on the pipeline without significantly increasing the execution time. In this part the execution is bound by latencies of each instruction. If the number of threads is increased above 256 the execution time of the kernel starts to increase significantly. In this part the maximum throughput of the pipeline is reached, so the execution time is throughput bound.

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 [11]. 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 [4]. 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 pipeline analysis step is detailed in Section IV-A. 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. These annotations enable a fast simulation of the pipeline behavior through execution on any OpenCL device (see Section IV-B). 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 core. These effects are incorporated into our model by the step called *Analytical Timing Model* in Figure 3. Details on this step are given in Section IV-C.

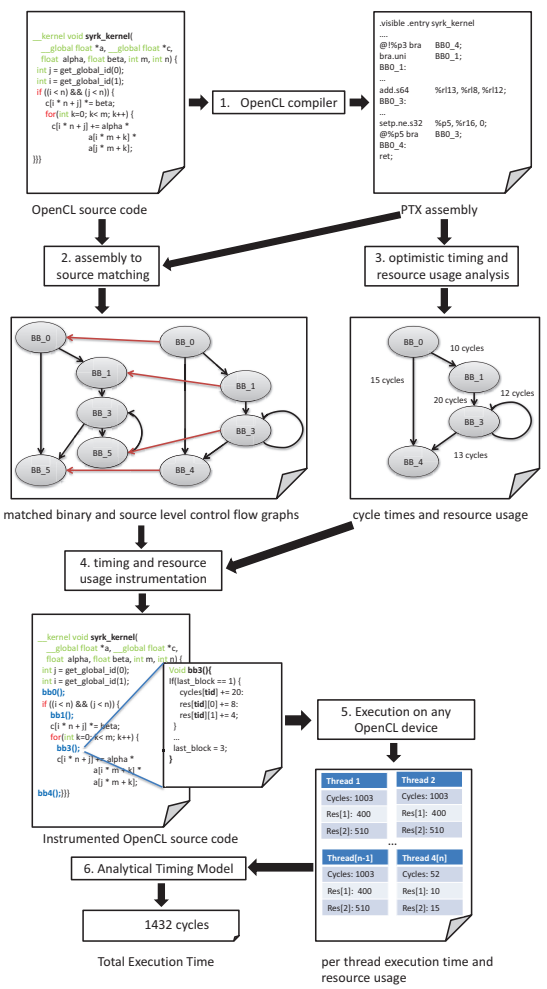


Fig. 3: Structure of the proposed simulation framework

A. *Basic Block Timing and Resource Analysis*

The static basic block timing analysis determines an optimistic execution time for each basic block in the binary level control flow graph. This analysis is run on the PTX assembly code and currently models the microarchitecture of a NVidia GTX 480 core. The analysis assumes that the pipeline is occupied by one warp exclusively. The basic block execution times generated by the static timing analysis are the basis for the timing simulation (Section IV-B). The effects of resource contention on the execution time are incorporated by an analytical model after the timing simulation (Section IV-C). The analysis uses pipeline execution graphs [12] to model the timing behavior of each *warp instruction* on the pipeline. Our pipeline execution graph EG_B for a basic block is defined as

$$EG_B = (S_B,\ D_B,\ lat,\ use,\ res)$$

where the nodes in S_B represents each execution step for each instruction on the pipeline. $D_B \subset S_B \times S_B$ represents the dependence relation. It contains an edge for each dependence corresponding to the instructions in the basic block. In our model an execution step might not directly correspond to a

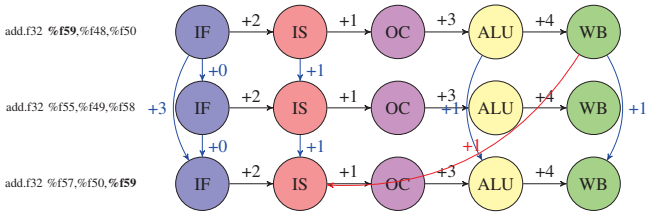


Fig. 4: Example of our pipeline analysis

pipeline stage. The minimum latency between the start of an execution step and the start of a dependent execution step is given by the function $lat: D_B \rightarrow \mathbb{N}_0$. Latency is expressed in cycles. The resource usage function $use: V_B \rightarrow \mathbb{N}_0$ labels each step in the execution graph by the number of cycles the resources in this step are taken. The function $res: S_B \rightarrow \mathbb{N}_0$ maps each execution step to a unique identifier for the resource used in this step.

As shown in Figure 1, the static analysis models each instruction's execution on the given pipeline as a graph with five nodes. The first node (IF) corresponds to the frontend part of the pipeline up to the instruction buffer. The second part (IS) models the issue stage and the scoreboard. The latency of register accesses in the operand collector units is modeled by the node labeled OC. The fourth node (ALU) models the timing effects of the actual execution. This node is labeled according to the used execution unit (ALU, SFU, MEM). The last node models the writeback stage of the pipeline. This node is labeled WB. Figure 4 shows an example of a pipeline execution graph for three add instructions on the pipeline of a GeForce GTX480. The latencies inherent to the pipelined execution of each instruction on the pipeline are shown by black edges. Instructions always take 2 cycles from instruction fetch to issue. The latency from issue to the operand collectors is one cycle for all instructions. The latency of the operand collectors depends on the number of registers read by the operation. The best case is always the number of registers read by the operation. The latency from execute to writeback can vary greatly depending on the instruction type. Load-/store instructions show one cycle under the assumption of a cache hit and full coalescing of memory accesses, while double precision floating point division has a latency of 330 cycles. Resource dependencies between instructions are modeled by the blue edges in Figure 4. As the frontend fetches two adjoining instructions at a time in program order each instruction is connected by an edge with latency zero in program order. To integrate the additional latency that every second instruction is only fetched when the instruction buffer is empty, we add additional edges between every second instruction with a latency of three cycles. The issue stage issues one instruction a cycle in program order, this is modeled by an edge with latency one between each successive instruction. Resource dependencies in the execute stage are modeled in the same way. If there are multiple copies of a resource, the modeled pipeline has two ALUs, the resource dependency edges skip instructions to account for the multiplicity of the resource. The same applies to the writeback step, as the pipeline can writeback two results at a time, there is only a dependency between every second node in (WB). Data dependencies are always modeled by an edge with

latency one from the writeback of the preceding instruction to the issue state of the depending instruction. Provided a pipeline execution graph, the timing analysis is done as a fixpoint iteration on the pipeline execution graph as shown in Algorithm 1.

```
Data: EG_B = (S_B, D_B, lat, use, res)
Result: t_B basic block execution time
t_B = 0
for s ∈ S_B do
    t_start(s) = 0
end
changed = True
while changed do
    changed = False
    for s ∈ S_B do
        for (p, s) ∈ D_B do
            if t_start(s) < t_start(p) + lat(p, s) then
                t_start(s) = t_start(p) + lat(p, s)
                t_B = max(t_B, t_start(s))
                changed = True
            end
        end
    end
end
```

Algorithm 1: Computing basic block cycle times

The algorithm iterates over each state in the pipeline execution graph and calculates the earliest start times for each node by the maximum over the start times of the predecessors added with the corresponding latencies. The fixpoint iteration is finished when the start times for each state do not change anymore. To accelerate the convergence of the fixpoint iteration we iterate over the states in topological sort order, but the result of the iteration is independent of the order in which the states are visited. The termination of the algorithm follows from the absence of cycles in the execution graph. For the static execution time analysis, we build the pipeline execution graph for the instructions of each basic block and calculate the fixpoint using Algorithm 1. It delivers a static timing analysis for the duration of the basic block by the maximum start time of each node in the execution graph. The analyzed execution time for a basic block v_i is called t_{v_i}. Building the execution time of a kernel during simulation by summation over the basic block times t_i would overestimate the execution time of the kernel, because the execution of instructions from adjacent basic blocks can overlap. To incorporate this effect into the static timings, we also build pipeline execution graphs for the instructions from each pair of adjacent basic blocks. The analyzed time for each pair of basic blocks v_i, v_j is called $t_{(v_i,v_j)}$. In the next section we show how the results of the static analysis are used to simulate the timing behavior of a kernel by executing a timing annotated version of the kernel's original source code.

B. Timing Annotation and Simulation

The timing information and resource information from the static timing analysis is annotated back in the original OpenCL source code, by inserting function calls to timing functions in the original source code, and generating the corresponding timing functions. The algorithm for timing annotation is shown in Algorithm 2. The algorithm first iterates over all mapped basic blocks in the source level control flow graph and inserts function calls that do the timing simulation according to the static analysis. The functions for timing simulation are generated in the inner loop of algorithm 2. In this loop the algorithm iterates over all binary level paths between matched blocks and calculates the estimated execution time for this path by the sum over the pairwise execution times of the nodes in the path $t_{(v_{i-1},v_i)}$. As the summation would count the execution time for each start node twice we subtract the execution time for each basic block $t_{v_{i-1}}$. In addition to the execution times we also annotate the threads resource usage for all resources in the pipeline. The results of an execution of the annotated source code on an OpenCL compatible device are an optimistic timing estimation of each thread and each the accumulated resource usage of the thread for each resource in the pipeline. The final execution time of the kernel is estimated from these values using an analytical model. The analytical model is explained in the next section.

```
Data: CFG_B = (V_B, E_B), CFG_S =
      (V_S, E_S), T_B, T_E, map
Result: Annotated version of CFG_s
for v_S ∈ V_S do
    if ∃v_B ∈ V_B : map(v_B) = v_S then
        insert function call to timing simulation in v_S
        create function for timing simulation
        for paths p between a mapped block v'_B and v_B
        do
            Add code to function for:
            if last simulated block = v'_B then
                t_thread += Σ_{v_i∈p/v'_B} t_(v_{i-1},v_i) − t_{v_{i-1}}
                for resources r_i do
                    u_thread(r_i) +=
                    Σ_{v_i∈p/v'_B} use(r_i, v_i)
                end
            end
        end
    end
end
```

Algorithm 2: Algorithm for timing annotation

C. Analytical Timing Model

The timing model uses the per thread execution times and resource usages as determined by the native execution of the annotated source code to calculate an estimate of the execution time of the whole kernel. The analytical model follows the hierarchy of parallelism of the multithreaded execution. The algorithm starts with the execution time of the threads as determined by the source level simulation. The execution time of a warp is then calculated from the execution time of the threads that form this warp. The execution times of all warps in a work group are combined to form the execution time of a work group. The execution time of the whole kernel is then estimated from the execution times of all work groups in the kernel. Timings up to this point do not consider the resource contention due to parallel execution of warps in the pipeline. This resource contention is taken into account by a final correction step.

The first step of the timing estimation calculates the

execution time of each warp. This is done by first determining which threads form a common warp and then calculating the execution time of a warp t_{W_i} by taking the maximum execution time of all threads in the warp.

$$t_{W_i} = \max_{t_i \in W_i}(t_{t_i})$$

This calculation is motivated by the fact that all threads in a warp are executing the same instruction in lockstep. Due to branch divergence it is still possible that the simulated execution time of the threads in warp show different execution times. The effects of branch divergence are approximated by taking the maximum execution time of all threads in the warp. This does not fully simulate all timing effects of branch divergence but accurately handles the most important case, of branch divergence at an *if* statement without an *else* or branch divergence at a loop exit condition. Real branch divergence in an if-else statement is not fully handled by our current model, but taking the maximum execution time still approximates the timing effects of this case. The execution times of a local work group WG_j is modeled by the end time of the last warp t_{last_j} in the work group. All warps in a local work group are started at the same time. So we can calculate the end time of the last warp by taking the maximum execution time of all warps in the local work group.

$$t_{last_j} = \max_{t_{w_i} \in WG_j}(t_{w_i})$$

Due to the limited bandwidth of the issue stage, the threads of a local work group the i-th warp of a local work group issues at least $\lfloor i/2 \rfloor$ cycles after the first thread of the local work group. To incorporate this in our model we add this to the finish time of the last warp. This leads us to the final equation for the execution time of the last warp

$$t_{last_j} = \max_{t_{w_i} \in WG_j}(t_{w_i} + \lfloor i/2 \rfloor)$$

The finish time of the last warp in each local work group is used to calculate the execution times of a kernel. The local work groups of a kernel can be executed in parallel but due to constraints of a GPU's hardware not all local work groups might be able to run in parallel. Given the number of parallel work groups n_{par}, we estimate the finish time of each work group. The first n_{par} work groups are started in parallel. The next work group is started when a work group finishes. This is expressed by the following equation:

$$t_{WG_j} = \begin{cases} t_{last_j} & : j < n_{par} \\ \min_{k \in \{j-n_{par},\ldots,j-1\}}(t_{WG_k}) + t_{last_j} & : j \geq n_{par} \end{cases}$$

The upper part of the equation calculates the finish times for the first n_{par} work groups, by using the finish time of the last warps. All further work groups are simulated by taking the minimum over the n_{par} predecessors and adding the finish time of the last warp in this work group. The optimistic execution time of the whole kernel is then the maximum finish time over all work groups:

$$t_{kernel\_opt} = \max_{WG_i}(t_{WG_i})$$

The kernel execution time so far does not consider any delays due to resource conflicts between multiple warps on the same pipeline. These resource conflicts are modeled by the last step of our analytical model. The analytical model first calculates the resource usage $use_{W_j}(r_i)$ of a warp W_j as the maximum resource usage over all threads in the warp:

$$use_{W_j}(r_i) = \max_{t_k \in W_j}(use_{t_k}(r_i))$$

The resource usage is calculated for each resource r_i. We then calculate the resource usage of the whole kernel by summation over the resource usage of all warps in the kernel:

$$use_{kernel}(r_i) = \sum_{W_i \in Warps} use_{W_i}(r_i)$$

The optimistic execution time is then combined with the kernels maximum resource usage to form the final execution time estimate. The most successful combination of resource usage and optimistic execution time we have found is the maximum of both values.

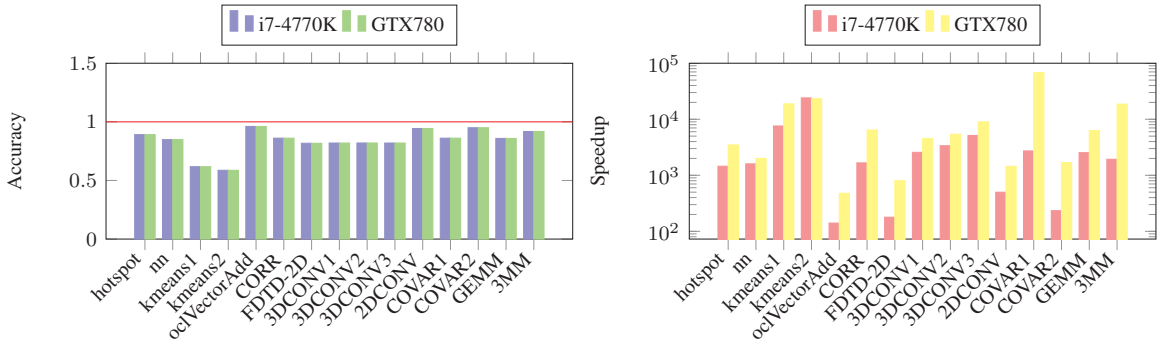
$$t_{kernel} = \max(t_{kernel\_opt}, \max_{r_i \in R} use_{kernel}(ri))$$

This model is surprising as it only takes resource contention into account when it is certain that there must be resource conflicts in the pipeline. But as GPUs are optimized for a high throughput this model seems a reasonable choice. More pessimistic models led to a severe overestimation of runtimes.

V. Results

We evaluated the performance model using several synthetic and real world benchmarks. All simulations were run on an Intel Core i7-4770K CPU at 3.5 GHz with a NVidia GTX780 GPU with 12 streaming multiprocessors at 952 MHz. For the execution of the instrumented source code we used either the CPU using Intel's implementation of OpenCL for CPUs or on the GPU using NVidias implementation of OpenCL for GPUs. For comparison we used the cycle accurate GPU simulator *gpgpu-sim*. Our configuration is based on the configuration for a NVidia GTX480 GPU as delivered by gpgpu-sim, but we reduced the model to include only one streaming multiprocessor. We also activated the so called perfect memory mode of gpgpu-sim. This mode handles all memory accesses as cache hits. We choose to use a simulator for our evaluation as only limited information on the internal architecture of actual GPUs is available, and we needed to verify the results of the pipeline model ignoring influences from the memory subsystem.

The proposed simulation framework has been implemented as a shared library implementing the runtime API for translation and running of OpenCL kernels. For performance simulations the library can be preloaded using the standard Unix `LD_PRELOAD` mechanism, so no changes to the host binaries used for simulation are needed. The kernels were translated to PTX code using *clang* as compiler [11]. All benchmarks were run with most compiler optimizations enabled (`-O3`) and we used the compiler switch to enable debug information in the compiler generated assembly files (`-g`). We evaluated the performance model with benchmarks from the Rodina [13] and polybench-gpu [14] Benchmarks. Figure 5a shows the execution times as estimated by our method divided by the execution times provided by `gpggpu-sim`. All our execution times underestimate the execution time as is expected by the best-case assumptions made by the performance modelling. For all but two kernels the proposed simulation technique provides an accuracy of 80% or higher.



(a) Accuracy

(b) Speedup

Fig. 5: Accuracy and speedup results for the kernels of standard benchmarks. In benchmarks containing multiple kernels the kernels are numbered by the order of appearance.

The accuracy results do not change between execution of the instrumented source code on a CPU or GPU as the performance simulation can be run on any OpenCL compatible device. The speedups in terms of the simulation time are shown in Figure 5b. The simulation time of our tool includes the time for data transfers from and to the OpenCL device the execution of the kernel on the device and the execution time of the analytical resource conflict model. As our goal is to support the simulation of long running application scenarios the speedups do not include the time used for the static analysis of GPU kernels and binary to source matching. When instrumented source code is run on a CPU speedups range between 140 for *oclVectorAdd* and 24061 for *kmeans2*. If the instrumented source Code is run on a GPU the speedups range between 477 for *oclVectorAdd* and 67750 for *COVAR*. The variation of execution speeds is partly explained by different basic block sizes in the applications. The other important factor considering speedups is the proportion of execution time of a thread to the number of threads. In our model, threads are simulated with the parallelism of the OpenCL device used for the simulations but our analytical performance model is run on the host without the use of parallelism. All benchmarks except *kmeans1* show an improvement of the simulation speed when the instrumented source code is executed on the GPU. The simulation performance of *kmeans1* degrades slightly as this benchmark does not fully utilize the available parallelism on the GPU.

VI. Conclusion and Future Work

In this paper we have presented a method for source level performance simulations of GPU cores that reaches high simulation speeds by doing the major work through execution of a timing annotated version of the applications source code. It is the only system that can simulate a GPUs execution time fast enough to make performance simulations on real world problems possible. In its current form the simulations deliver a best case approximation of the performance of a GPU. The best-case approximations are useful for first performance simulation on large application scenarios. In our future work we are going to increase the simulation accuracy by incorporating missing features like modeling accesses to registers more accurately and to extend the model to simulate diverging branches and cached memory accesses. A model for the memory subsystem will also allow us to compare the method to excution time measurements on a real GPU. We will also integrate our approach into a system to do high performance simulations of heterogeneous architectures including GPUs.

VII. Acknowledgments

This work was 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., 2014

[2] K. Lu, D. Muller-Gritschneder, and U. Schlichtmann. Hierarchical control flow matching for source-level simulation of embedded software. *SOC '12*, 2012.

[3] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Fast and accurate source-level simulation of software timing considering complex code optimizations. *DAC '11*, 2011.

[4] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Dominator homomorphism based code matching for source-level simulation of embedded software. *CODES+ISSS '11*, 2011.

[5] Z. Wang and A. Herkersdorf. An efficient approach for system-level timing simulation of compiler-optimized embedded software. *DAC '09*, 2009.

[6] A. Gerstlauer, et al. Abstract System-Level Models for Early Performance and Power Exploration. *ASP-DAC '12*, 2012.

[7] A. Bakhoda, et al. Analyzing CUDA workloads using a detailed GPU simulator. *ISPASS '09*, 2009.

[8] J. Lai and A. Seznec. Break down GPU execution time with an analytical method. *RAPIDO '12*, 2012.

[9] A. K. Parakh, M. Balakrishnan, and K. Paul. Performance Estimation of GPUs with Cache. *IPDPS '12*, 2012.

[10] A. Betts and A. Donaldson. Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis. *ECRTS '13*, 2013.

[11] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *CGO'04*, 2004.

[12] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems 29*, June 2006.

[13] S. Che, et al. Rodinia: A benchmark suite for heterogeneous computing. *IISWC '09*, 2009.

[14] S. Grauer-Gray, et al. Auto-tuning a high-level language targeted to GPU codes. *InPar '12*, 2012.