ΣVP: Host-GPU Multiplexing for Efficient Simulation of Multiple Embedded GPUs on Virtual Platforms

YoungHoon Jung and Luca P. Carloni
Dept. of Computer Science
Columbia University, New York, NY
{jung,luca}@cs.columbia.edu

ABSTRACT

Despite their proliferation across many embedded platforms, GPUs present still many challenges to embedded-system designers. In particular, GPU-optimized software actually slows down the execution of embedded applications on system simulators. This problem is worse for concurrent simulations of multiple instances of embedded devices equipped with GPUs. To address this challenge, we present ΣVP, a framework to accelerate concurrent simulations of multiple virtual platforms by leveraging the physical GPUs present on the host machine. ΣVP multiplexes the host GPUs to speed up the concurrent simulations without requiring any change to the original GPU-optimized application code. With ΣVP, GPU applications run more than 600 times faster than GPU-software emulation on virtual platforms. We also propose Kernel Interleaving and Kernel Coalescing, two techniques that further speed up the simulation by one order of magnitude. Finally, we show how ΣVP supports simulation-based functional validation and performance/power estimation.

1. INTRODUCTION

Since their advent in 1999, Graphics Processing Units (GPUs) have progressively benefited the performance of many computing systems with their specialized parallel architectures. Originally designed to serve on desktop computers, nowadays GPUs play an important role in a variety of systems. Since the introduction of the use of GPUs for general-purpose computing (GPGPU) a growing number of high-performance computing systems have adopted them [16]. GPGPU has found its way also into mobile and embedded systems for a variety of applications, including sensor-data processing and computer vision [6, 21]. Furthermore, these systems are increasingly integrated in large-scale networks to form distributed embedded systems and support such applications as multiplayer online gaming [5, 15].

Given these trends, designers are increasingly interested in simulating the execution of GPU applications on the computing systems that they are designing and that will host one or more GPUs. Simulation with multiple instances of virtual platforms (VPs) enables many important design decisions as part of the process of exploring the design space of the *target systems* [9, 10, 11]. Since this process requires the simulation of complex application scenarios, the speed of the simulator is of critical importance.

DAC '15, June 07 - 11 2015, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2744769.2744913.

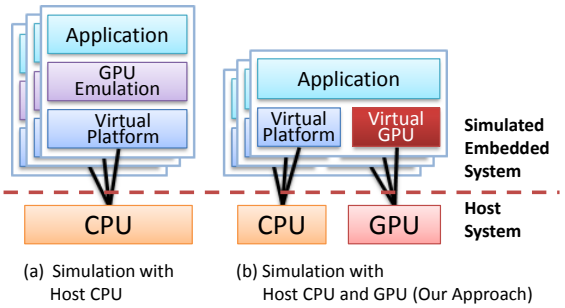


Fig. 1: Two ways of simulating GPU applications.

However, using GPU-optimized code in a simulation environment presents some challenges. While it accelerates a given application on the target system, **the addition of GPU-specialized software code can slow down the simulation** of the application execution. The reason is that most of the current multi-node system simulators run the entire simulation on the *host CPU*. Then, in order to run the GPU code, many simulators, and even widely-adopted development tools such as the Android Emulator, need to include GPU emulation capabilities (e.g. the Mesa software backend) [1, 18]. The presence of an additional software layer on top of the VP significantly deteriorates the overall execution speed [8, 20]. Fig. 1(a) illustrates this scenario of simulating embedded applications that have GPU code by using GPU emulation on top of a VP that runs on the host CPU. In contrast, we propose to take advantage of the increasing presence of physical GPUs in many host systems. As shown in Fig. 1(b), the idea is to execute the GPU code from multiple virtual GPU models on the host GPU.

To demonstrate this idea we developed (ΣVP) - *Simulation using GPU-Multiplexing for Acceleration of Virtual Platforms*, a framework to simulate embedded devices equipped with embedded GPUs. Our system executes separately the target CPU code on the simulated CPU and the target GPU code on the simulated GPU, thus enabling a modular integrated simulation of multiple embedded systems.

ΣVP benefits from two novel optimization techniques, Kernel Interleaving and Kernel Coalescing, that we developed thanks to the possibility of executing multiple VP instances on virtual embedded GPUs. We show that ΣVP can be used

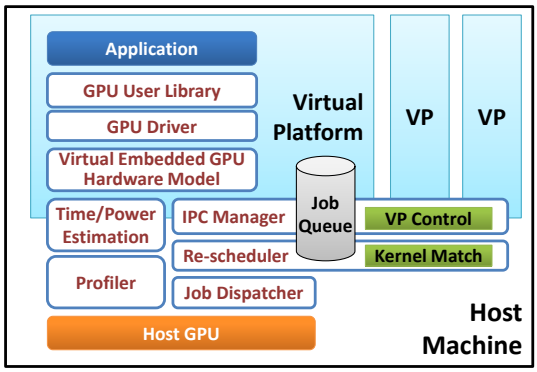


Fig. 2: Proposed simulation framework prototype.

for functional validation, timing analysis and estimation of power dissipation. Our approach not only speeds up the simulation time by orders of magnitude but it also enables major savings in terms of the efforts to build the models for these timing and power analyses.

2. GPU MULTIPLEXING FOR SIMULATION

ΣVP multiplexes the host GPUs to execute the request from the VPs by using separate streams for each VP. Thanks to our methods for time-division multiplexing (interleaved invocations) and throughput-division multiplexing (coalesced invocations), the host GPUs can be used to accelerate the execution of the target GPU code. In this section, we present the components of ΣVP and how they interact.

The Architecture of ΣVP. Fig. 2 shows the structure of the prototype that we developed to evaluate our ideas. ΣVP supports many VP instances, each consisting of three main modules: a GPU user library, a GPU driver, and a virtual embedded GPU hardware model. We designed these to efficiently resolve the three challenges discussed above.

The **GPU User Library** forms a layer that intercepts the requests from user applications by providing the same APIs of the physical GPUs, e.g. the CUDA runtime library. We designed the user library for the virtual GPU model to support binary compatibility with existing GPU applications. Hence, the application binaries that use GPU instructions do not need any change to run on the virtual GPUs. Instead, the user library forwards the requests from those applications to the virtual **GPU device driver**. This is a driver for the guest operating system that works as an interface between the GPU user library and the virtual GPU hardware model. Finally, the **Virtual Embedded GPU Hardware Model** pushes the requested kernels into the *Job Queue* in the host machine through the IPC manager.

On the host machine, there are five modules that run on top of the physical GPU. The **Inter-Process Communication (IPC) Manager** allows the virtual embedded GPUs and the host GPU to communicate through an IPC method such as socket or shared memory. Inside the IPC manager, there is a submodule, named VP control, that stops and resumes the VPs to support the Kernel Interleaving optimization technique for synchronous kernel invocations, which is presented in Section 3. The **Re-scheduler** has two functions. First, it reorders the asynchronous kernel jobs in the Job Queue by keeping a partial order in the original VP. It is a non-preemptive, optimal scheduler augmented for job dependencies [14]. Second, it combines identical kernel requests in the Job Queue into one single kernel job, by using Kernel Coalescing, also discussed in Section 3. The **Job Dispatcher** links the requests to the GPU driver library on the host machine and invokes the physical GPU instructions based on the requests in the *Job Queue*. The **Time/Power Estimation** module estimates the execution time and the power consumption on the target GPU, while we actually execute the kernel on the host GPU based on the profiling information, as described in Section 4. Finally, the **Profiler**, which is provided by the manufacturer, acquires execution information such as the number of executed instructions (per instruction type), the elapsed clock cycles, and the percentages of each occurred stall.

3. TWO OPTIMIZATION TECHNIQUES

Kernel Interleaving and Kernel Coalescing are two techniques that we developed to improve the performance of simulating the execution of GPU commands from different applications on multiple virtual-platform instances.

Kernel Interleaving. GPU architectures feature two types of engines that can operate in parallel: a Compute Engine and a Copy Engine. Although some recent GPUs support *Concurrent Kernel Execution* that may automatically interleave kernels from distinct streams, this can lead to suboptimal performance, as shown in Fig. 3(a). Kernel Interleaving reorders the executions to reduce the wasted cycles across the two engines and improve the overall execution time by using the expected time for each invocation, as shown in Fig. 3(b). To implement Kernel Interleaving we followed two distinct approaches for the two kernel-invocation types that are supported by GPUs: synchronous and asynchronous. To effectively interleave instructions from different programs, ΣVP reorders the asynchronous requests in the *Job Queue* as shown in Fig. 4(a). For synchronous kernel invocations, instead, ΣVP cannot fetch the next GPU instructions until it finishes the current one. However, since it is possible to control the progression of the execution of each VP, we can stop one for some time to let another one run. This property can be used to perform kernel interleaving for synchronous GPU calls, as shown in Fig. 4(b).

Coalescing Identical Kernels. Generally, the invocation of a function in a program suffers from some overhead: the program must backup and restore the register values, deliver the function arguments, jump to the function code, and finally return to the main program. In many cases of embedded applications, it is important to reduce such overhead. We observed that when multiple VP instances are running it is likely that an identical kernel is called by more than one VP at the same time [19]. Such simulations can be accelerated by coalescing those common invocations from each VP into a single kernel invocation. ΣVP makes this possible through an appropriate management of memory. When kernel coalescing is necessary, ΣVP first coalesces the memory chunks into one bigger piece of data stored at physically-contiguous memory locations, as shown in Fig. 5. Then, the GPU can run one kernel instance to process the merged data set.[1] After the kernel execution, the resulting data are properly divided to be copied from the GPU device back to the host memory addresses.

Fig. 6 illustrates this idea for the case of two kernel instances: instead of executing them as shown in Fig. 6(a),

[1]Merging memory chunks for kernel coalescing is different from global memory access coalescing [12].

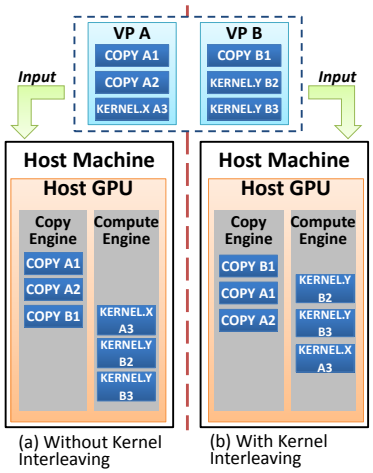


Fig. 3: Kernel Interleaving.

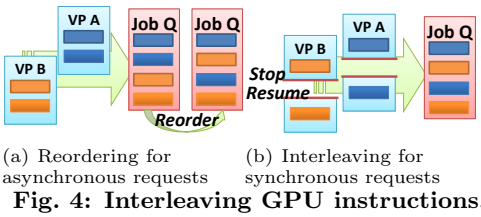


Fig. 4: Interleaving GPU instructions.

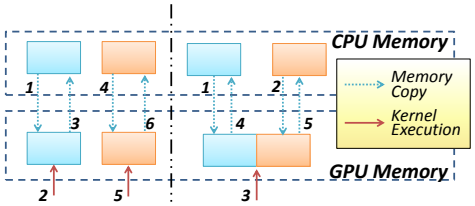


Fig. 5: Coalescing two memory chunks (left) into one (right).

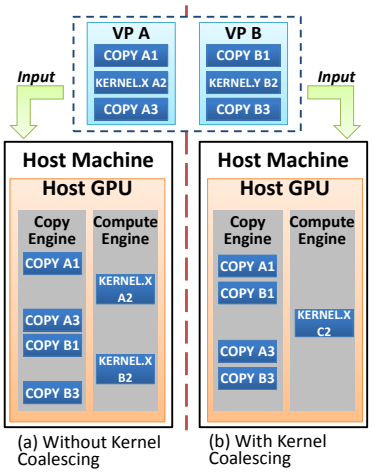


Fig. 6: Kernel Coalescing.

Kernel Coalescing allows us to execute a single kernel instance on a larger data set as shown in Fig. 6(b).

This technique brings another significant gain: data alignment. Due to their parallel architecture, GPUs are designed to execute multiple concurrent threads. Hence, whenever the data size is not aligned, the GPU must run another loop of the kernel for the rest of the data. This handicap can be significantly reduced by coalescing memory chunks.

4. TIME AND POWER ESTIMATION

To augment ΣVP with capabilities for timing and power analysis we developed *Profile-Based Execution Analysis*, a novel method that combines the information obtained executing the kernel on the host GPU with the information obtained compiling it for the target GPU and with existing models for time and power estimation [7, 13].

Fig. 7 illustrates the main idea of this method. First, ΣVP compiles the kernel for both the target and the host GPU architectures. Second, ΣVP executes the kernel on the host GPU and gathers a variety of kernel-profiling information from this execution including: number of executed instructions for each instruction types (floating point and integer arithmetic, control flow, and memory access), elapsed clock cycles, cache hit/miss counts, and stall reasons. Then, ΣVP derives various execution profiles as if the kernel was executed on the target GPU. For instance, by combining the iteration count[2] and the number of instructions of each program block[3], ΣVP derives the expected instruction count $\sigma_{\{K,T\}}$ for the kernel K executed on the target GPU T, as shown in Fig. 8. The same method can be extended to obtain σ for each instruction type i:

$$\sigma_{\{K,T\}} = \sum_i \sum_b \left[\lambda_b \cdot \mu_{\{b_i,T\}} \right] \quad (1)$$

where $b \in K$ is a program block; $i \in \{\text{FP32, FP64, Int, Bit, B, Ld, St}\}$ is an instruction type; $\mu_{\{b,T\}}$ is the static number of instructions from b compiled for T; and λ_b is the iteration count of a block b in the execution.

[2]The iteration count can be estimated via several probabilistic methods [4]. For more precise evaluation, we dynamically inserted PTX instructions into the kernel before the execution to obtain the iteration count. This involves less than 0.5% overhead.

[3]The largest portion of the kernel that has a distant execution path determined by control instructions.

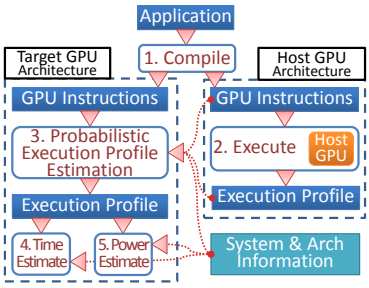


Fig. 7: Profile-based execution analysis.

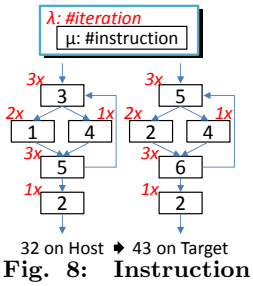


Fig. 8: Instruction count derivation.

Timing Estimation. We built three increasingly-refined models to estimate the number $C_{\{K,T\}}$ of clock cycles needed to execute a kernel K on the target GPU T. The first model is simply based on $\text{IPC}_{H\to T}$ that is the ratio of IPC_T and IPC_H, which are the maximum values of the number of Instructions Per Cycle on the target (simulated) and the host (simulating) GPU architectures, respectively. Then, we obtain our first estimate $C_{\{K,T\}}$ simply as:

$$C_{\{K,T\}} = \frac{\sigma_{\{K,T\}}}{IPC_H \times IPC_{H\to T}} \quad (2)$$

This, however, does not capture the characteristics of each GPU microarchitecture which may have an important impact (e.g. the same instruction may take different clock cycles on different GPUs or a smaller cache size can cause more memory access stalls). To better estimate the IPC_T we can use a probabilistic approach based on the execution latency τ of each instruction type i [13]. Since the ideal number of clock cycles spent on the host (excluding stalls) is given by:

$$C^{\mathfrak{P}}_{\{K,H\}} = \sum_i \left[\sigma_{\{K_i,H\}} \times \tau_{\{i,H\}} \right] \quad (3)$$

A second estimate of $C_{\{K,T\}}$ is:

$$C'_{\{K,T\}} = C^{\mathfrak{P}}_{\{K,T\}} + C_{\{K,H\}} - C^{\mathfrak{P}}_{\{K,H\}} \quad (4)$$

But this uses the exact stall delays occurred on H, which can lower the estimation accuracy. By augmenting our model with a probabilistic model of the data-cache behavior for data-dependency stalls [17], we get the third estimate:

$$C''_{\{K,T\}} = C'_{\{K,T\}} - \Upsilon^{[data]}_{\{K,H\}} + \Upsilon^{[data]}_{\{K,T\}} \quad (5)$$

Language	Executed by	Time (ms)	Ratio
CUDA	GPU	170.79	1.00
CUDA	Emul. on CPU	9141.51	53.52
CUDA	Emul. on VP	374534.34	2192.95
CUDA	**This work**	568.12	3.32
C	CPU	8213.09	48.09
C	VP	269874.03	1580.15

Table 1: Execution time of matrix multiplication.

where $\Upsilon^{[data]}_{\{K,H\}}$ are the data-dependency stalls occurred during the execution of K on H, calculated combining the probabilistic data-cache behavior model and the details of the host GPU architecture (e.g. the main memory size, the cache size and associativity).[4]

Power Estimation. Existing power models are based on the number of executed instructions per each instruction type [7]. We use a power-estimation method based on the calculated execution time and the expected execution profile on the target GPU. By combining the power consumption values for each instruction type and the static power dissipation $P^{[static]}_T$, which we empirically acquired, we estimate the power consumption during the execution of K on T as

$$P_{\{K,T\}} = P^{[static]}_T + \sum_i \left[\frac{\sigma_{\{K_i,T\}}}{ET_{\{K,T\}}} \times RP\_Component_{\{i,T\}} \right] \quad (6)$$

where $RP\_Component_{\{i,T\}}$ denotes the runtime power consumption dissipated by the microarchitecture components of T to execute the instruction of type i. The estimated execution time $ET_{\{K,T\}}$ is calculated as the estimated clock cycles divided by the product of the number of used GPU processors and the GPU clock frequency. We use C'' as the clock cycles for calculating the estimated power consumption.

5. EXPERIMENTAL RESULTS

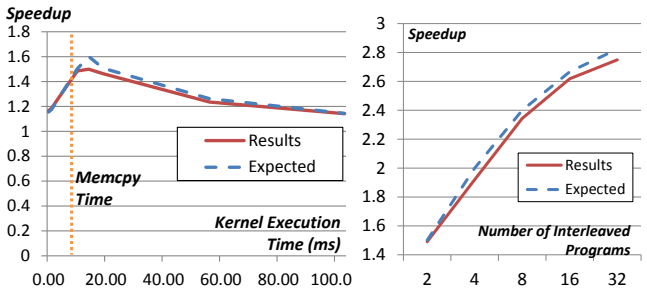
In this section we present a comprehensive set of experimental results that demonstrate the effectiveness of each of the methods described in the previous section.

Experimental Setup. Our proposed techniques can potentially be applied to various GPU programming platforms including OpenCL and OpenACC. In this paper, however, we demonstrate our method using CUDA for two main reasons: 1) we plan to extend our method to other CUDA related SDKs such as PhysX, a physics engine and 2) OpenCL is disabled on many recent Android devices.

We used a 32 Intel Xeon CPU machine with a NVIDIA Quadro 4000 GPU as the host environment and a QEMU ARM Versatile PB model as the target simulator. For the experiments of time and power estimation, we used also NVIDIA Grid K520 as another host GPU.

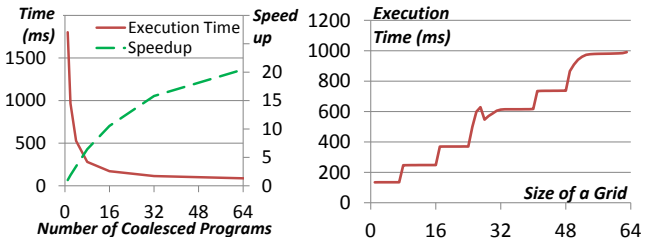
Leveraging host GPU. Our first experiment was a comparative evaluation of the different options to execute an embedded GPU application. For this we used a simple program that multiplies 300 times two 320×320 matrices of double-precision numbers. Table 1 reports the results for two versions of the program: a CUDA implementation (first 3 rows) and a C implementation (last 2 rows.) The first row, which corresponds to the native execution of the CUDA program on a GPU, is used as the baseline for the comparison. The execution of the CUDA program takes 53.52 times longer when running on a GPU emulator on top of a CPU and 2200 times longer when running on an ARM CPU model inside a VP through binary translation. Clearly, emulating GPU code inside a virtualized emulation model yields sub-optimal results. Nevertheless, this is still a common practice in many simulation frameworks of commercial products, which, for instance, use the MESA open-source libraries to run OpenGL ES applications [8]. In fact, as shown by the last two rows of Table 1, running the C version of this program on either the CPU or the VP is faster than running the CUDA program on a GPU emulator inside a VP. In contrast, our proposed GPU multiplexing technique is only 3.32 times slower than native execution.

[4]Some GPU manufacturers provide the details of their product architectures while some studies discovered the information by microbenchmarking [22].



(a) Speedups for Kernel Lengths (b) Speedups for VP Numbers

Fig. 9: Experiments for Kernel Interleaving.



(a) Coalescence Effectiveness (b) Impact of Data Size

Fig. 10: Experiments for Kernel Coalescing.

Kernel Interleaving. We consider two interleaved GPU programs, each with a loop that iterates: a memory copy from host to device, a kernel execution, and a memory copy from device to host. Fig. 9(a) shows the speedups measured as varying the complexity of the kernel while keeping the size of the input data constant. The time for memory copy is 13.44 ms, represented as a vertical orange dotted line.

Kernel Interleaving can shorten the total execution time from $3N$ instructions to $2 + N$ instructions, where N is the number of programs to be interleaved, under the assumption that each instruction takes about the same amount of time. If the kernel execution time T_k and memory copy time [5] T_m are different, the total time is given by:

$$T_{total} = 2T_m + N \cdot Max(T_m, T_k) \quad (7)$$

which is represented by the blue line in Fig. 9(a). The red line shows the actually measured experimental values, which are quite close to the expected values. This experiment confirms that the highest speedup through Kernel Interleaving is obtained when the kernel execution time is similar to the memory copy time (indicated by the orange dotted line). This is a form of latency hiding.

[5]This means the time for memory copies before the kernel execution; e.g. matrix multiplication needs two input memory copies.

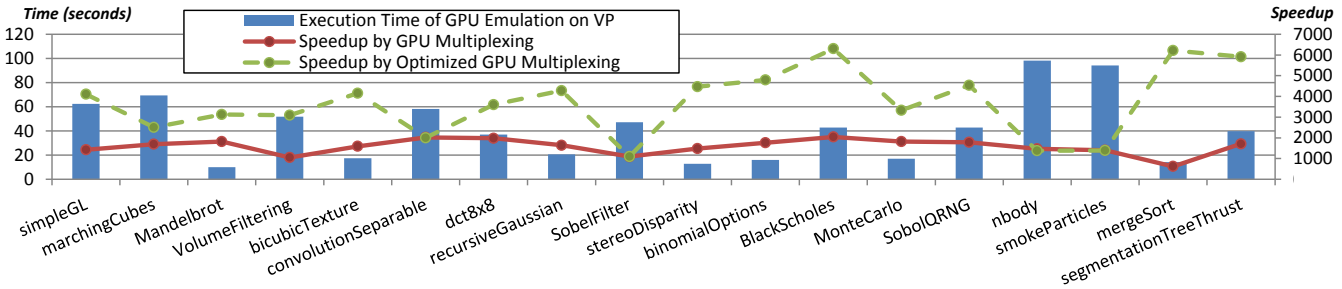


Fig. 11: Experimental results: GPU-VP emulation vs ΣVP with optimizations.

While the previous experiment is for two interleaved programs, Fig. 9(b) shows the speedups as function of N interleaved programs, from 2 to 32. Since the execution time without Kernel Interleaving is $3T$ when $T_k = T_m = T$, the speedup is expected to grow with N as:

$$Speedup = \frac{3 \cdot N \cdot T}{(2+N) \cdot T} = \frac{3N}{2+N} \tag{8}$$

which is represented by the blue line in Fig. 9(b). For large number of interleaved programs the speedup is about 3×.

Kernel Coalescing. Fig. 10(a) shows the speed of executing *vectorAdd* as function of the number of GPU programs to coalesce. The total size of the input vectors remains the same across the different numbers of programs. In other words, the same amount of work is distributed over the given number of programs. The solid red line indicates the total execution time of the coalesced program and the green dashed line indicates the speedup of the same horizontal coordination, with the result of no coalescing (one program) being the comparison base. For instance, when coalescing 16 GPU programs the time to complete the executions of the applications is 171 ms, for a 10.54X speedup. These results confirm that Kernel Coalescing can indeed reduce the execution time. The speedup reaches 20.48 times for the case of 64 programs. A large portion of the gain can be attributed to the impact of data-size alignment given the number of concurrent threads, the unit of computation the GPU can simultaneously hold. In CUDA the number of concurrent threads used for a kernel is decided by the size of a block (a group of threads) and the size of a grid (a group of blocks). A kernel is executed by a grid of thread blocks. Fig. 10(b) shows the execution time of one single kernel as the size of the data grows and, accordingly, the size of a grid increases from 1 to 64 (while the number of threads in a block remains 512). The resulting curve roughly resembles a staircase, which implies that a kernel execution with an unaligned grid size wastes some portion of its resources. For instance, the same execution time is obtained both for a grid of size 9 and a grid of size 16 even though the data sizes to be processed are different, being $9 \times 512 = 4608$ and $16 \times 512 = 8192$ data units, respectively. For a given size of a grid the expected execution time is

$$T_{expect} = T_o + T_e \times \lceil \xi_{input} / \lambda \rceil \tag{9}$$

where T_o is the overhead time spent for launching kernels, T_e is the kernel execution time for the alignment unit size of data, ξ_{input} is the size of the input data, and λ is the aligned unit for the GPU's processing ability.

In summary, these preliminary experiments confirm the effectiveness of the two optimization techniques that we developed in ΣVP so that they can be automatically applied to the simulation of embedded GPU programs on VPs.

Performance Comparison. Here we present a complete evaluation of our simulation framework using the suite of benchmark GPU applications available as part of the CUDA SDK [2]. In particular, we compared the simulation of these applications on the VPs for three scenarios: 1) GPU emulation on the VP; 2) simulation on the host GPU with our proposed GPU multiplexing; and 3) simulation on the host GPU with our GPU multiplexing plus the two optimization techniques: Kernel Interleaving and Kernel Coalescing.

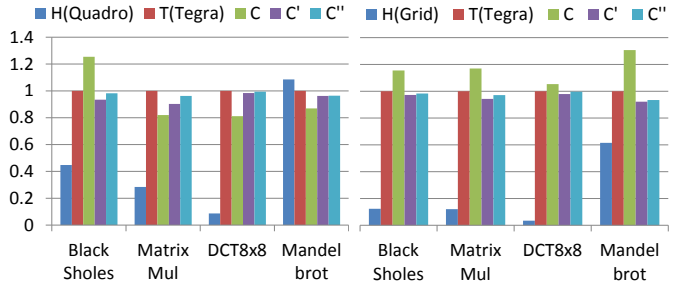
Fig. 11 reports the experimental results. The blue bar shows the execution time of emulating the GPU applications concurrently on eight VP instances. For example, when each of these executes *simpleGL*, the time for completing all the executions is about 62 seconds. The green dashed line and the red solid line indicate the speedup achieved by the host GPU multiplexing with and without the two proposed optimization techniques, respectively. Thus, for *simpleGL* GPU multiplexing provides a simulation speedup of 1428 (with respect to the blue bar), while the addition of the two optimizations achieves a speedup equal to 4104.

The analysis of the red solid line suggests that applications that use less floating-point instructions, e.g. *VolumeFilter*, *SobelFilter*, *stereoDisparity*, and *mergeSort*, have relatively lower speedups than others. Also, some non-CUDA operations (e.g. file operations or OpenGL invocations) limit the speedups for *Mandelbrot*, *bicubicTexture*, *recursiveGaussian*, *MonteCarlo*, and *segmentationTreeThrust*, which read from input files or write to output files, as well as *simpleGL*, *marchingCubes*, *VolumeFiltering*, *SobelFilter*, *nbody*, and *smokeParticles*, which use OpenGL for graphics. The reason is that these portions of the applications are not the target of the acceleration provided by ΣVP.

The analysis of the green dashed line confirms that the effect of the two optimization techniques varies across the applications based on their use of CUDA instructions. Applications such as *convolutionSeparable*, *dct8x8*, *SobelFilter*, *MonteCarlo*, *nbody*, and *smokeParticles*, have kernels that are not sped up by the two optimizations, mostly due to the way they access and manage the memory. All remaining applications benefit from the optimizations.

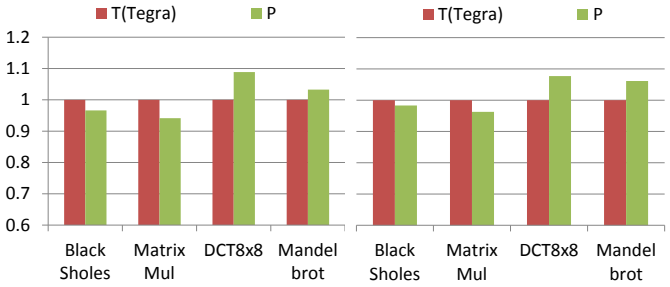
The speedup obtained with GPU multiplexing varies from 622 times (*mergeSort*) to 2045 times (*BlackScholes*) compared to the emulated GPU on the VP. The speedup with both GPU multiplexing and the two optimizations varies between 1098 times (*SobelFilter*) and 6304 times (*BlackScholes*). In the best case (*mergeSort*) the addition of the two optimizations yields an additional 10X speedup.

Timing Estimation. We evaluated the accuracy of our timing estimation models as follows. The estimated time values are calculated for the target GPU (NVIDIA Tegra



(a) Estimates using Quadro 4000 (b) Estimates using Grid K520

Fig. 12: Normalized execution times: two observations on target and host GPUs and three estimates.



(a) Estimates using Quadro 4000 (b) Estimates using Grid K520

Fig. 13: Normalized power dissipation: an observation on target GPU and an estimate $P_{\{K,T\}}$.

K1) and normalized by the observed execution time on an actual target GPU. We experiment with execution profile from two different host GPUs, NVIDIA Quadro 4000 and Grid K520. Fig. 12 shows the measured execution times on the target GPU and the host GPU, and the three expected execution times $ET_{\{K,T\}}$ based on C, C', and C'', respectively. As expected, the execution times observed on the host GPU are much shorter than the observed and estimated values for the target GPU. On the other hand, the results demonstrate that the estimated execution times are close to the measured values from a real target device. The fact that the estimates are close to 1 no matter which host GPU is used for execution profile confirms that our models work well across the different host GPU architectures.

Power Estimation. The results of Fig. 13 compare the estimated power dissipation with the one measured on the actual device. Our estimations are within about 10% of the actual values, thus confirming that ΣVP can be effectively used also for simulation-driven power analysis.

6. RELATED WORK

A recent Android platform's experimental patch brings the OpenGL ES 2.0 instructions from the emulator to the host OS, converts to standard OpenGL 2.0, and runs natively on the host GPU [1]. This approach, however, works only for OpenGL in a single emulator. Many GPU simulators are based on software models for GPUs [3]. Some timing estimation methods use these software models to obtain execution traces [13]. These approaches run very slow, while ΣVP offers effective ways to estimate execution time and power consumption in addition to fast simulation.

7. CONCLUSIONS

We proposed a technique to efficiently simulate multiple instances of virtual platforms that run GPU applications. Compared to the emulation of GPUs on VPs, the speed of our simulation framework is between 1000 and 6000 times faster when running a large set of GPU applications. We achieved this major improvement by leveraging the presence of GPUs on the host systems and by optimizing the execution of GPU kernels with two novel optimization methods: Kernel Interleaving and Kernel Coalescing. Further, by presenting a novel estimation method that leverages the execution of a kernel on the host GPU, we showed how our framework can be used not only for full-system simulation but also for timing analysis and power estimation.

Acknowledgments. This work is partially supported by the NSF (A#: 1219001), and by C-FAR (Contract #: 2013-MA-2384), one of the six SRC STARnet centers.

8. REFERENCES

[1] Android (`developer.android.com`).
[2] CUDA (`developer.nvidia.com/cuda`).
[3] S. Collange et al. Barra: A parallel functional simulator for GPGPU. In *Proc. of MASCOTS*, pages 351–360, Aug. 2010.
[4] L. David and I. Puaut. Static determination of probabilistic execution times. In *Proc. of ECRTS*, pages 223–230, June 2004.
[5] M. M. Hassan, H. S. Albakr, and H. Al-Dossari. A cloud-assisted IoT framework for pervasive healthcare in smart city environment. In *Proc. of EMASC*, pages 9–13, Nov. 2014.
[6] J. Hensley, J. Isidoro, and A. J. Preetham. Combining computer vision and physics simulations using GPGPU. In *SIGGRAPH Sketches*, pages 1–1, Aug. 2007.
[7] S. Hong and H. Kim. An integrated GPU power and performance model. In *Proc. of ISCA*, pages 280–289, June 2010.
[8] Y. Joo, D. Lee, and Y. I. Eom. Delegating OpenGL commands to host for hardware support in virtualized environments. In *Proc. of ICUIMC*, pages 95:1–95:4, Jan. 2014.
[9] Y. Jung and L. P. Carloni. Cloud-aided design for distributed embedded systems. *IEEE Design & Test*, 31(4):32–40, Jul-Aug 2014.
[10] Y. Jung, J. Park, M. Petracca, and L. P. Carloni. netShip: a networked virtual platform for large-scale heterogeneous distributed embedded systems. In *Proc. of DAC*, pages 169:1–169:10, June 2013.
[11] T. Kempf et al. A SW performance estimation framework for early system-level-design using fine-grained instrumentation. In *Proc. of the Conf. on DATE*, pages 468–473, Mar. 2006.
[12] Y. Kim and A. Shrivastava. Memory performance estimation of CUDA programs. *ACM TECS*, 13(2):21:1–21:22, Sept. 2013.
[13] J. Lai and A. Seznec. Break down GPU execution time with an analytical method. In *Proc. of RAPIDO*, pages 33–39, Jan. 2012.
[14] M. Lombardi, M. Milano, and L. Benini. Robust non-preemptive hard real-time scheduling for clustered multicore platforms. In *Proc. of DATE*, pages 803–808, Apr. 2009.
[15] S. C. McLoone, P. J. Walsh, and T. E. Ward. An enhanced dead reckoning model for physics-aware multiplayer computer games. In *Proc. of DS-RT*, pages 111–117, May 2012.
[16] J. Owens et al. GPU computing. *Proc. of the IEEE*, 96(5):879–899, May 2008.
[17] V. Puranik, T. Mitra, and Y. N. Srikant. Probabilistic modeling of data cache behavior. In *Proc. of EMSOFT*, pages 255–264, Oct. 2009.
[18] J. W. Sheaffer, D. Luebke, and K. Skadron. A flexible simulation framework for graphics architectures. In *Proc of HWWS*, pages 85–94, Aug. 2004.
[19] G. Shen et al. Accelerate video decoding with generic GPU. *IEEE CSVT*, 15(5):685–693, May 2005.
[20] R. Ubal et al. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proc. of PACT*, pages 335–344, Sept. 2012.
[21] Y.-C. Wang, S. Pang, and K.-T. Cheng. A GPU-accelerated face annotation system for smartphones. In *Proc. of MM*, pages 1667–1668, Oct. 2010.
[22] H. Wong et al. Demystifying GPU microarchitecture through microbenchmarking. In *Proc. of ISPASS*, pages 235–246, Mar. 2010.