GPU Performance Prediction Through Parallel Discrete Event Simulation and Common Sense

Guillaume Chapuis
Los Alamos National Laboratory
Bikini Atoll Rd., SM 30
Los Alamos, NM 87545, USA
gchapuis@lanl.gov

Stephan Eidenbenz
Los Alamos National Laboratory
Bikini Atoll Rd., SM 30
Los Alamos, NM 87545, USA
eidenben@lanl.gov

Nandakishore Santhi
Los Alamos National Laboratory
Bikini Atoll Rd., SM 30
Los Alamos, NM 87545, USA
nsanthi@lanl.gov

ABSTRACT
We present the GPU Module of a Performance Prediction Toolkit developed at Los Alamos National Laboratory, which enables code developers to efficiently test novel algorithmic ideas particularly for large-scale computational physics codes. The GPU Module is a heavily-parameterized model of the GPU hardware that takes as input a sequence of abstracted instructions that the user provides as a representation of the application or can also be read in from the GPU intermediate representation PTX format.

These instructions are then executed in a discrete event simulation framework of the entire computing infrastructure that can include multi-GPU and also multi-node components as typically found in high performance computing applications. Our GPU Module aims at a trade-off between the cycle-accuracy of GPU simulators and the fast execution times of analytical models. This trade-off is achieved by simulating at cycle level only a portion of the computations and using this partial runtime to analytically predict the total execution of the modeled application.

We present GPU models that we validate against three different benchmark applications that cover the range from bandwidth- to cycle-limited. Our runtime predictions are within an error of 20%. We then predict performance of a next-generation GPU (Nvidia's Pascal) for the same benchmark applications.

Categories and Subject Descriptors
C.0 [**Computer Systems Organization**]: Modeling of computer architecture; I.6.5 [**Computing Methodology**]: Simulation and Modeling—*Model Development*

General Terms
Performance, Experimentation, Measurement

Keywords
Parallel Discrete Event Simulation, GPGPU, Performance Prediction

1. INTRODUCTION
GPUs have become popular in the high-performance computing (HPC) domain as commodity hardware, where GPU-based clusters usually take at least some of the top spots in the Top500 (see www.top500.org) list of most powerful super-computers. GPU-based clusters are also projected to be among most powerful next generation super computers [17]. Predicting performance of next generation GPUs relevant to the actual application portfolio that HPC users have, such as national research laboratories, with an emphasis on computational physics codes, is a key challenge for performance prediction. The US Department of Energy, with its traditional need for high performance computing, is executing a strategy of hardware/software co-design, which at times requires significant code refactoring in order to take advantage of novel architectures, such as GPU-based clusters. The Performance Prediction Toolkit (PPT) is a DOE co-design project that develops a comprehensive prediction capability of how computational physics codes and methods perform on novel hardware architectures, thus enabling a fast adoption of new code by quickly identifying and ruling out unsuccessful refactoring scheme.

PPT models both hardware and software at levels of abstraction that are appropriate to the concrete question at hand applying a mix of discrete event simulation, stochastic models, and analytical closed-form expression at various layers of the software and hardware stacks. For instance, modeling at a relatively high level, we have used PPT to evaluate new computational physics methods in the area of accelerated molecular dynamics [10]. The performance of these physics methods depended on the quite involved statistical properties of the physics system and the corresponding loop structure of the code on the software side; on the other hand a fairly simple model of the underlying hardware that included the number of cores, clock speed and interconnect speed as main parameters was sufficient for the study.

Other applications and computational physics kernels, which often perform algebraic operations, such as matrix operations, require a much more detailed model of the available hardware because data motion, efficient cache hierarchy usage, and latency hiding largely determine performance.

VALUETOOLS 2015, December 14-16, Berlin, Germany
Copyright © 2016 ICST
DOI 10.4108/eai.14-12-2015.2262575

Compared to a CPU design, a GPU architecture is simpler to model. The relative shallowness of its hardware and software stacks allows for a more predictable behavior. The absence of features such as instruction reordering, branch prediction and deep cache hierarchy - not to mention the absence of a full fledged operating system running at the same time -, make execution of code on a GPU much easier to model than on a modern CPU.

In this paper, we present the GPU Module of the Performance Prediction Toolkit. Our GPU model consists of a large set of parameters. It is embedded in the parallel discrete event simulation engine Simian [12] written in Python. It interacts with application models through an interface that we call a task list. Upon execution, the GPU model predicts the time it takes to execute the task list. The task list itself can be generated manually, through a Python script, or directly from the intermediate representation PTX format that is commonly used for GPU code. The loop-structure of an application can be wrapped around the task list computations, such that we achieve realistic run time predictions not just of individual GPU operations but rather of entire applications.

We present models for three GPUs: a Tesla M2090, a K40, and a Quadro K6000. We validate these models against three benchmark applications, namely, an optimized matrix multiplication, a naive Jacobi stencil implementation, and an optimized Jacobi stencil implementation (all taken from the Parboil benchmark [14]), that collectively cover the relevant spectrum of computational physics applications ranging from bandwidth- to cycle-limited. Our runtime predictions are within an error of 20%. We then predict performance of a next-generation GPU (Nvidia's Pascal) for the same benchmark applications. Our model predicts that Pascal generation GPUs will benefit bandwidth- and cycle-limited applications by a factor of up to 2.5 over Kepler based GPUs on application from our benchmark.

The paper is organized as follows: after describing related work in Section 2, Section 3 describes our GPU Module and also includes a brief introduction to GPU architectures. Section 4 describes our validation results, Section 5 describes our prediction results for future GPU architectures.

2. RELATED WORK

Modeling and emulating the runtime behavior of GPUs is an active field of research. When it comes to predicting the performance of a piece of code on a given GPU architecture, many analytical approaches have been proposed [8, 6, 9, 13]. In [8], the authors propose an analytical model of a GTX280 GPU, which combines and adapts to GPUs existing models for parallel computations and memory accesses. The analytical model presented in [6] addresses memory-level parallelism and thread-level parallelism behaviors of GPUs and also concerns a GTX280 GPU. In [9], the authors use the performance models from [6] and focus on the effect of code optimization using code skeletons as input across two generations of Nvidia GPUs. In [13], the authors refine and improve the model proposed in [6] to identify bottlenecks in GPU code. Some efforts have also been made towards modeling specific components of a GPU. In [11], the authors focus on modeling the caching behavior of GPUs using the reuse distance obtained from an execution trace. In [1], the authors focused on the performance of choleski factorization on heterogeneous clusters and thus proposed a detailed model of CPU GPU communications developed using Simgrid [3].

On the other end of the spectrum, many GPU emulators have been proposed [5, 4, 2, 15]. These tools fully model the run-time behavior of GPUs and can be used to obtain various metrics on a given execution without even having access to the GPU. Authors in [5] propose an emulator called Ocelot, which allows the execution of Cuda code on a multicore CPU through binary translation. In [4], the authors describe Barra, a GPU emulator, which can run GPU code on multicore CPUs and mimics the behavior of the GPU in order to provide insights into how a given piece of code would be executed. GPGPU-sim [2], emulates the run time execution of a cuda code (using the intermediate representation as input) to predict the performance on a given GPU and gain information about potential bottlenecks. Finally, multi2sim [15] combines low-level models of an x86 CPU and an AMD GPU to predict the performance and instrument hybrid code.

Analytical models present the advantage of fast simulations but lack the accuracy and bottleneck finding capabilities of emulators. Emulators on the other hand exceed in runtime the execution of the code on the actual GPU and therefore cannot be used to explore optimization strategies for pieces of code with very long execution times. We propose a hybrid approach, which uses a cycle-level emulation of a subset of the computations and extrapolates the result of this emulation to obtain the predicted total execution time.

3. MODELING GPUS

In this section, we use Nvidia Cuda's terminology to briefly describe the architecture of GPUs. We then describe the methodology we used to model executions on GPUs.

3.1 GPU architecture

The architecture of a GPU marks a radical change with respect to the traditional design of CPUs. In this architecture, high processor frequencies, large caches and low latencies are traded for a higher bandwidth and a higher number of processing units. This somewhat new paradigm provides higher computational throughput and thus better performances for highly parallel problems at the expense of programmability.

GPUs are composed of a number of streaming multiprocessors (SMX). These multiprocessors share access to an off-chip memory space, later referred to as global memory - see Figure 1. Access to global memory by the SMXs presents a high latency - several hundred cycles - and a high bandwidth - hundreds of GB per second. This global memory can also be accessed by the CPU and is used to send input data and read results back as computations on the GPU are always performed at the request of the CPU.

Each SMX contains various computational resources: single and double precision units (SPUs and DPUs respectively), special function units (SFUs) and load/store units (LD/ST). SPUs - respectively DPUs - handle single - respectively double - precision operations on floats and integers; SFUs han-

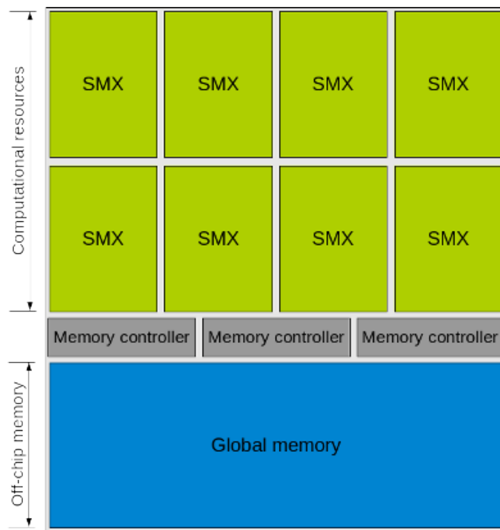

Figure 1: Overview a GPU's architecture. The chip contains several streaming multiprocessors. A large but slow, off-chip memory is accessible to the computing units.

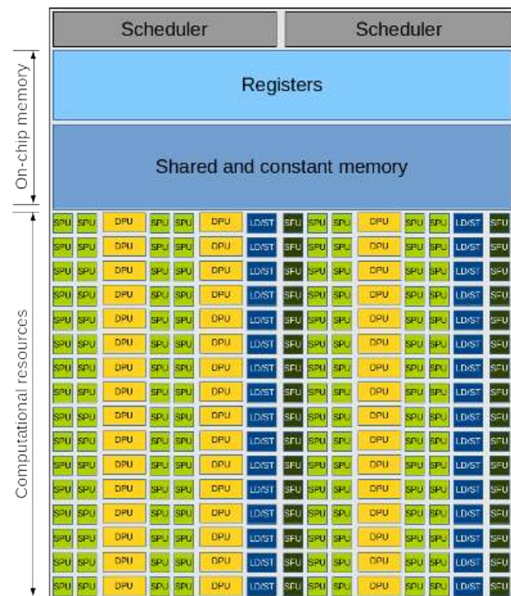

Figure 2: Overview a streaming multiprocessor (SMX). Each SMX contains schedulers to issue instructions, computational resources (single precision unit, double precision units etc.).

dle operations such as exponentiation, square roots, sines and cosines; LD/ST handle address translations for memory accesses. Each SMX also contains a small amount of low-latency on chip memory that can be used as a cache and for inter thread communication - see Figure 2. Compared to a CPU core, a greater percentage of the chip is devoted to computational resources as opposed to fast memory and control flow logic.

Each SMX can be seen as a processor with rather low clock frequency when compared to a traditional CPU core. These processors, however only compute instructions on 1024 bit vectors - or 32 times a four-byte float - which is 32 times larger than most CPU instructions and 4 times larger than the largest available SIMD instruction vectors in current CPUs. This difference allows for a greater computational power for highly data parallel problems, ie. problems that can exploit large vector instructions for most operations. In order to exploit the large vector operations, problems to be run on a GPU are decomposed into a computation grid. The grid is composed of blocks of groups of 32 threads called warps each executing the same sequence of instructions in a lock-step fashion - see Figure 3. Due to this fact, each warp can be seen as a single thread executing large vector instructions and the thread-level granularity can most of the times be ignored. Warps from the same block are executed on a single SMX and can communicate with each other using the fast on-chip memory. All warps from the same grid share the same sequence of instructions but can be at different levels of completion, whereas threads of the same warp execute the same instruction at any given time. Unlike with CPUs, there is no instruction reordering done on a GPU, which means that instructions are always executed by all warps in the order provided by the sequence. However, an instruction can be issued before the previous instruction is completed if there is no dependency.

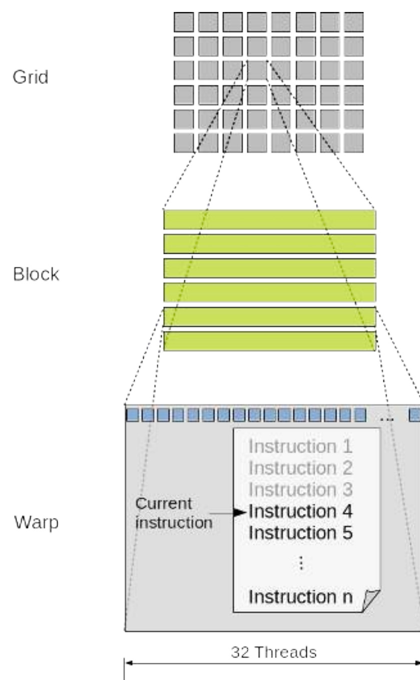

Figure 3: Example of a CUDA grid. The grid is composed of blocks; each block is composed of warps (or groups of 32 threads). Threads in a warp execute instructions in a locked-step fashion.

Blocks of the grid are scheduled to the SMXs until one of various conditions is reached. These conditions are availability of register space, on-chip memory space or a hardware limitation on the number of concurrent blocks or warps on a single SMX. Each SMX thus has a pool of warps to issue instructions to. Warps for which the current instruction depends on the results of a previous instruction that has not completed are called inactive. At a given clock cycle, active warps can potentially be issued an instruction. Each SMX contains a certain number of warp schedulers, which try to issue one, or more if independent, instructions to a given active warp at every clock cycle. The issuing of an instruction however depends on the availability of the required computational resources.

Once a block is completed - ie. when all its warps have completed the last instruction in the sequence - a new block can be attributed to the SMX if there are more blocks available in the grid. Different types of bottlenecks can occur; if warps mostly stall on memory accesses, the problem is said to be memory bound. For a given memory bound problem, if the bandwidth limit is almost reached, it is said to be bandwidth bound; if the bandwidth limit is not reached and memory stalls still occur predominantly, the problem is latency bound. In rarer cases, a problem can be limited by the availability of computational resources; it is then said to be compute bound. Achieving performance on GPUs relies on a deep understanding of the underlying architecture, whereas decades of software and hardware development on CPUs provide programmers with a much better abstraction of the hardware. From a modeling point of view, the relative simplicity of the hardware and software stack of GPUs, renders their behavior much easier to understand and predict. Recent trends in CPU and GPU designs, however, show that both architectures are slowly merging to a common design, with the Intel MIC being a perfect example of an architecture half-way between a traditional multicore CPU and a manycore GPU.

3.2 Simulation methodology

In order to model GPUs, we aim at a trade-off between the cycle accuracy of emulators and the fast simulation runtimes of analytical methods. To achieve this, we take advantage of the simplicity of the block-scheduling scheme implemented in GPU hardware to simulate at cycle-level only a portion of the execution of the grid and use the obtained partial runtime to analytically estimate the total execution time of the grid. Computations on a GPU are represented as a grid of blocks. These blocks are distributed to available SMXs until a maximum occupancy per SMX is reached; this problem- and hardware-dependent maximum number of blocks is later referred to as *max_block*. These blocks are executed using parallelism and pseudo-parallelism - using time slicing techniques - on a given SMX. Once a block is completed, a new one, if available, is issued by the block scheduler to the SMX. We model at cycle-level the completion of *max_block* blocks on a single SMX. By taking advantage of the regularity of this block scheduling method, the time taken to complete these blocks is then used to analytically predict the total runtime of the grid - see Figure 4.

The entire grid is therefore divided into workloads of *max_block* blocks. When computing the runtime of a single workload

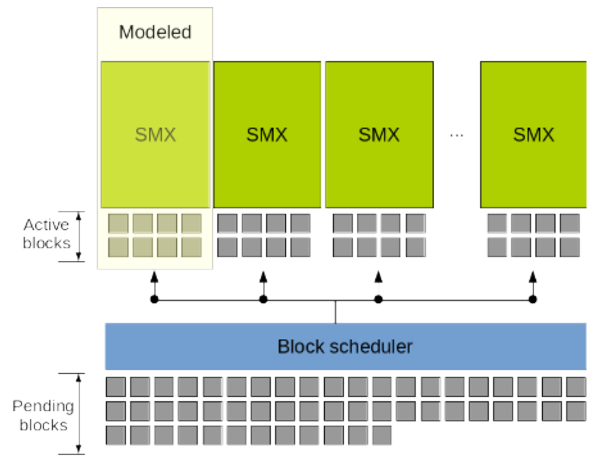

Figure 4: Blocks are scheduled to available SMXs until the maximum number of concurrent blocks is reached on each SMX. We model the execution of this maximum number of concurrent blocks on a single SMX and extrapolate the runtime for the computation of the entire grid.

on an SMX, we simulate every cycle and attempt to issue instructions to available warps until the completion of the instruction sequence of each warp. At each cycle, a number of warps can be issued an instruction up to the number of warp scheduler available on the modeled GPU. If the GPU has dual-issue capabilities, we attempt to issue two instructions to warps. A warp is active if dependencies for its next instruction have been met and an instruction can be issued if computational resources are available. It would be wrong to assume that all blocks in a workload will complete at the same time, which is why we take into account the completion time of each block in a workload to compute the potential overlap between workloads. In order to obtain reliable predictions for small grids - ie. grids with a number of blocks close to the number of available SMXs -, we need to determine how the block scheduler works in such cases. Microbenchmarking results show that blocks are distributed evenly to SMXs instead of fully occupying the first SMXs. This behavior might however change in future Cuda releases as GPUs will tend to be used more as computation servers with concurrent applications and kernels running on the same GPU. Achieving high performance depends heavily on hiding the latency of global memory accesses. Hiding the memory latency is mainly done using instruction-level parallelism and thread-level parallelism. Instruction-level parallelism occurs when two (or more) consecutive instructions are independent; in such cases, the second instruction can be issued before the completion of the first one (provided computational resources required for this second instruction are available). Thread-level parallelism occurs when an SMX issues instructions to different warps, thus hiding the latency-induced stalling of inactive warps. Our low-level model of the completion of a workload can simulate both these types of latency-hiding mechanisms.

We model an application by defining a task list which represents the sequence of instructions of the code to simulate. Items of this list can be of the following type: single preci-

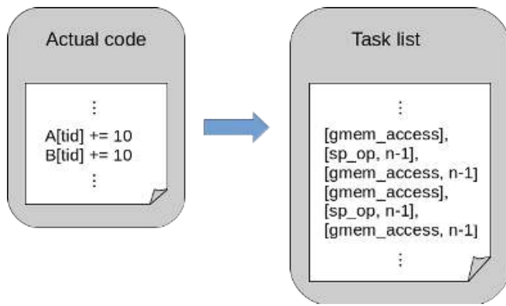

Figure 5: Example of a conversion from real code to the task list representation. This two lines of code (left) translate to six tasks in the task list (right).

sion operation, double precision operation, special function operations, block synchronization and memory access (along with the type of memory space being accessed, global, shared or constant). Each operation in the task list is followed by the index of the operations (if any) on the completion of which it depends. We chose this simple representation of applications to easily allow the modeling of applications which have not been written yet. When Cuda code for the application exists, the task list is automatically generated from the PTX instruction sequence. Dependencies between instructions are easily obtained using register names. The current absence of instruction reordering on GPUs ensures that the overall structure of the application, described in the task list, is followed at runtime on an actual GPU.

Figure 5 shows an example of conversion of existing code to our task list representation. This example assumes that arrays A and B reside in global memory. Adding ten to an element of these arrays yields three tasks for each array: loading the value of the element from global memory, computing a single precision operation and writing the result back to global memory. In this case, the second and third tasks depend on the completion of the previous task; therefore, no instruction-level parallelism can be achieved. However, tasks 3 and 4 are independent accesses to global memory. If resources are available (ie. LD/ ST units and bandwidth), task 4 can be issued before the completion of task 3. The first and fourth tasks of this example depend on the computation of the value of variable *tid*; for simplicity, this dependency is however not shown in this example.

Using PTX code as input does not provide the same level of accuracy as using binary level information but provides with a GPU independent representation of the code; meaning that a single task list is required to simulate executions on various GPUs. This task list can be edited to predict the effect of code modifications on various architectures. This simple task list representation, however, does not take into account actual memory addresses being accessed by the simulated program. Therefore, memory address-dependent phenomena such as partition clamping, shared memory bank conflicts and caching behavior cannot be modeled. Including memory address information would make the task list representation much more complicated, thus rendering the modeling of future applications too cumbersome. Caching behavior could nevertheless be approximated using information such as average reuse distance while allowing the task list representation to remain simple.

4. VALIDATION

In this section, we describe our validation benchmark as well as our modeled GPUs. We then present the results of our predicted performances on the validation benchmark.

4.1 Description of the benchmark

Our validation benchmark is composed of three code examples taken from the Parboil benchmark [14]. The first example, is an optimized matrix multiplication inspired by the one from [16]. This piece of code makes use of the fast on-chip memory to cache reused values and relieve bandwidth usage. Despite these efforts, the optimized code is still mostly bandwidth bound for large matrix sizes and latency bound for small matrices. This example is later referred to as SGEMM (for single precision general matrix multiply). Instances for this piece code are composed of square matrices with dimension ranging from 412 to 16384.

Our second example from the Parboil benchmark is a naive implementation of a Jacobi stencil operation over a 3D grid. This naive implementation does not make use of the fast on-chip memory cache but global memory is nevertheless accessed optimally - ie. all accesses are coalesced. This implementation of the problem is highly bandwidth bound even for small instances. This example is later referred to as STENCIL. We run this code over volumes of size ranging from 16 to 512 in all three dimensions.

Our last benchmark example is an optimized implementation of the Jacobi stencil operation over a 3D grid. Unlike our previous example, this implementation make heavy use of the fast on-chip memory and is optimized for large instances. The use of this local cache greatly reduces the bandwidth requirements making this piece of code mostly compute bound. This example is later referred to as OPT_STENCIL. For this optimized implementation, volume sizes range from 16 to 1024 in all three dimensions. These applications were automatically translated to PPT task lists using the PTX code as input.

4.2 Modeled GPUs

We model the three types of GPUs that we have at our disposal: the M2090 from the Fermi generation, and the Tesla K40 and Quadro K6000 from the Kepler generation. Among those three, the older M2090 is the least powerful in almost every aspect: clock frequency, amount of computational resources, bandwidth etc.The number of SMX, however, has remained rather stable across the two generations with 16 SMX for the Fermi GPU and 15 SMX for the Kepler ones. The M2090 has two warp schedulers per SMX, whereas the Kepler GPUs with much more computational resources have four warp schedulers. It is important to note that the kepler warp schedulers can dual issue instructions to a given warp in the presence of two independent and ready-to-compute instructions. The K40 and K6000 are based on the same chip design (GK110) and thus have very similar specifications. The main difference between the two being the higher clock frequency of the K6000 - 901.5MHz compared to 745MHz for the K40. An important difference between the two generations is the number of concurrent blocks that can run on

Name	M2090	K40	K6000
SMXs	16	15	15
SPUs (per SMX)	32	192	192
DPUs (per SMX)	16	64	64
SFUs (per SMX)	4	32	32
Schedulers (per SMX)	2	4	4
Dual issue	No	Yes	Yes
Max blocks per SMX	8	16	16
Max threads per SMX	1536	2048	2048
Frequency (MHz)	650	745	901.5
Bandwidth (GB/s)	177	288	288

Table 1: Overview of the specifications of the three modeled GPUs.

a single SMX. The M2090 is hardware limited to 8 blocks or 48 warps, when Kepler cards can go up to 16 blocks or 64 warps. The larger register file of the Kepler cards also means that register pressure is less of an issue.

Table 1 shows a subset of the parameters needed to create a model of a GPU. The information in this table is available in Nvidia white-papers and general documentation. Additional information such as latency and throughput of arithmetic and memory operations is obtained through microbenchmarks derived from [18], which we adapted to the 64 bit memory addresses of new GPU generations. Given this information, a new GPU model can be implemented in a matter of minutes by setting parameters accordingly in PPT .

4.3 Validation results

Measured runtimes, in this section, are given in seconds and include data transfers from the host to the GPU, a thousand runs of the kernel and transfers back to the host. Cuda code was compiled using nvcc flags to minimize the caching of global memory data, since our model does not yet take cache behavior into account. Figure 6 shows the measured and predicted performances for our benchmark on the M2090 GPU. Predicted runtimes for the SGEMM and the OPT_STENCIL examples are underestimated up to 20% for the 512 instance of the OPT_STENCIL. The 1024 instance exceeds the 6GB of global memory available on the M2090 and could therefore not be run; the memory required for this instance is two representations of a 1024*1024*1024 volume composed of floats (>8.5GB). Performances for the naive stencil implementation are however slightly over-estimated.

Results of our validation benchmark on the K40 GPU can be seen in figure 7. Performance prediction for the SGEMM and the naive stencil over-estimate the measure runtimes, while predictions for the OPT_STENCIL are under-estimated up to a roughly 20% error for the last instance.

Figure 8 shows the results of our validation benchmark for the K6000 GPU. Our predictions for the SGEMM closely match the actual runtimes, while predictions for the STENCIL and OPT_STENCIL examples are respectively over- and under-estimated. While some errors remain in our predictions, we can see that the global trends of all three examples are fairly well predicted with all modeled GPUs.

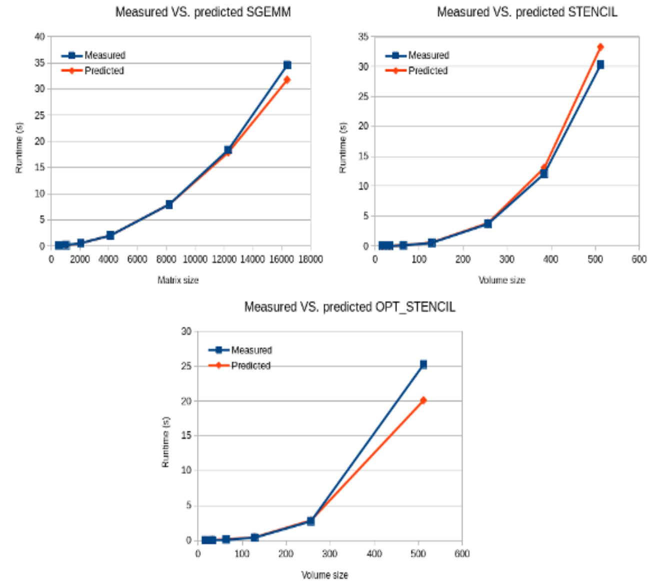

Figure 6: Actual and predicted performance for the M2090 GPU.

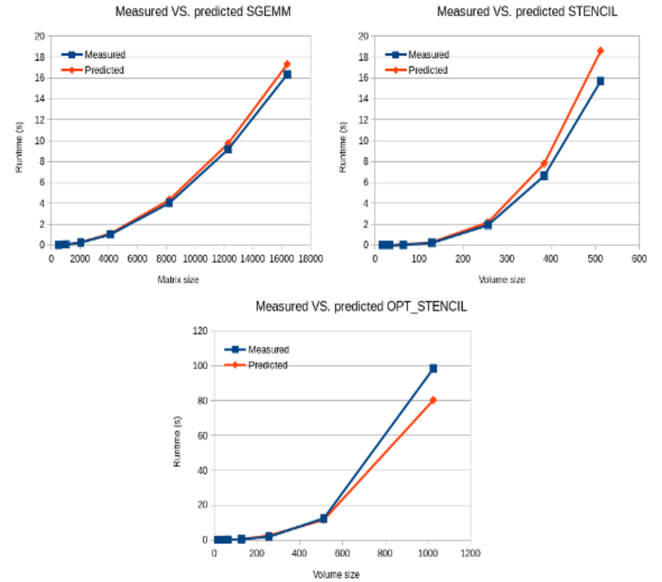

Figure 7: Actual and predicted performance for the K40 GPU.

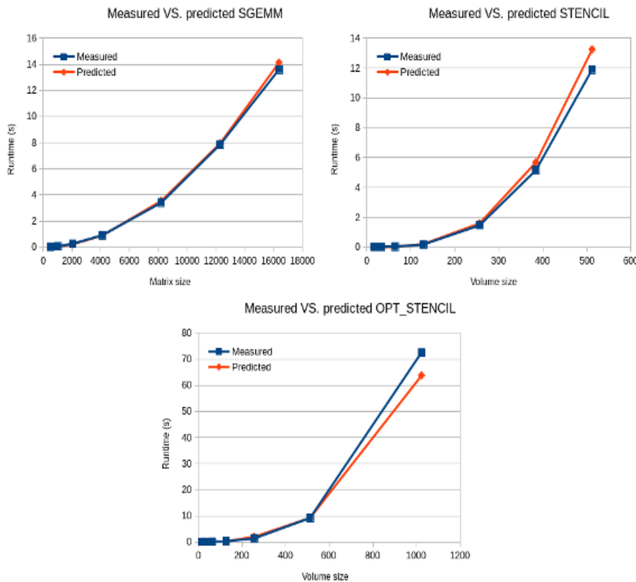

Figure 8: Actual and predicted performance for the K6000 GPU.

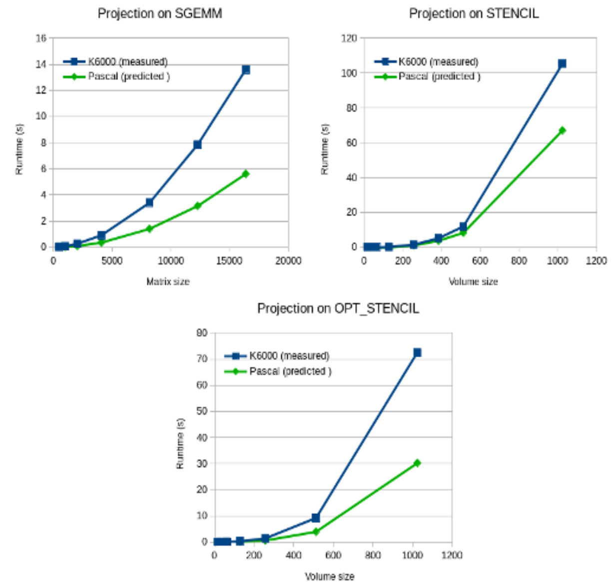

Figure 9: Projection of performances of a hypothetical Pascal GPU. Measured performances of the K6000 GPU are given as a reference.

5. RESULTS

In order to illustrate how the PPT GPU module can be used, we gather the scarce information about the future Pascal generation of Nvidia GPU and create a model of such a GPU. We first give the details of the parameters used to build the model and then apply it to predict the performance of this hypothetical GPU on our benchmark applications.

5.1 Future GPUs

Nvidia recently announced its new generation of GPUs called Pascal [7]. Since it has not been released yet, information about the architecture is still very scarce. Our model is therefore based on suppositions, which we describe here. This new generation will have a new type of 3D memory delivering a much higher bandwidth up to almost 1000 GB/s for higher-end cards. The global memory will also be physically closer to the chip than in previous generations, potentially yielding lower latency of memory accesses. Pascal generation GPUs will also have a faster connection to the CPUs memory, which Nvidia claims will be 5 to 12 times faster than current generation interconnects. This should lead to faster data movement between the host and the device. None of our benchmark applications, however, relies on a fast connection to the host, since we time a single pair of transfers back and forth with a thousand runs of the kernel. Though taken into account in our model, this new feature should therefore not impact our predicted performances significantly.

No information has yet been given about the number of SMXs nor the content of these future multiprocessors. Given the apparent stability across previous generations of the number of SMXs, we project this number to remain around 16. The recent trend seems however in an increased computational capacity per SMX. We therefore project 256 single precision units per SMX and up to 128 double precision units, as opposed to 192 and 64 respectively for the K6000 and K40 from the Kepler generation. Due to power consumption constraints and in accordance with recent trends, we project that processor clock frequencies will remain stable around 900 MHz for both Quadro and Tesla cards.

5.2 Predicted performances

Figure 9 shows our performance projections for the hypothetical Pascal GPU described previously. Actual performances obtained on the K6000 GPU are given as a reference. Projected performances for the Pascal GPU show a significant speedup of nearly 2.5x on SGEMM and OPT_STENCIL. The projected speedup for the naive stencil is however more modest with nearly 1.5x. These projected speedups do not get close to the 10x speedup over Maxwell GPUs claimed by Nvidia. Our benchmark codes, however, do not make use of the large amount of 16 bit precision floating point computational power introduced in future Pascal GPUs; moreover, our benchmarks do not heavily rely on communication between the host and the device, which will be greatly improved in Pascal GPUs. It is very likely that applications with such properties will see a higher speedup on the Pascal generation of GPUs.

6. CONCLUSION

We presented the GPU Module of a Performance Prediction Toolkit developed at Los Alamos National Laboratory. The aim of this toolkit is to quickly explore the hardware-software space to find the best match between algorithms and computer architectures for a given problem. This toolkit can help software developers in finding the best algorithmic variation for a given architecture or finding the best architecture to run a given piece of code before purchasing the machine. It also aims at testing algorithmic ideas on various architectures before even writing the code.

We described the modeling strategy used for our GPU model, which offers a trade-off between the cycle-accuracy of GPU emulators and fast execution of analytical models. This trade-off is achieved by modeling at the cycle-level only a portion of the computations and using the relative simplicity of the GPU architecture to use this partial runtime to analytically infer the total execution time of the entire program. Three recent Nvidia GPUs are modeled and validated against a subset of applications from the Parboil benchmark [14]. Our models showed a maximum error of around 20% and were successful in determining the global trends on all three benchmark applications. We then used available information to create a model of a GPU from the future Pascal generation and predicted its performance on our benchmark applications.

We plan on improving our GPU Module by adding a cache model to increase the accuracy of its prediction. Such a model for cache behavior could be based on reuse distance information so as to preserve the simplicity of application models and avoid having to include memory address-dependent information. New features will also be added to help software developers find bottlenecks in their applications using the insight provided by the cycle-level part of our model. We also intend to create models for GPU-cluster-size applications to test the validity of our entire Performance Prediction Toolkit. Such applications will include multi-GPU computations across a cluster of hybrid machines.

7. REFERENCES

[1] E. Agullo, O. Beaumont, L. Eyraud-Dubois, J. Herrmann, S. Kumar, L. Marchal, and S. Thibault. Bridging the gap between performance and bounds of cholesky factorization on heterogeneous platforms. In *Heterogeneity in Computing Workshop 2015*, 2015.

[2] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[3] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. Simgrid: a sustained effort for the versatile simulation of large scale distributed systems. *arXiv preprint arXiv:1309.1630*, 2013.

[4] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A parallel functional simulator for gpgpu. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010*, pages 351–360. IEEE, 2010.

[5] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364. ACM, 2010.

[6] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[7] J.-H. Huang. Nvidia gpu roadmap, 3 2014. Keynote speech by Nvidia CEO Jen-Hsun Huang at the Annual GPU Technology Conference in San Jose, Calif.

[8] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. Narayanan, and K. Srinathan. A performance prediction model for the cuda gpgpu platform. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 463–472. IEEE, 2009.

[9] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. Grophecy: Gpu performance projection from cpu code skeletons. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 14. ACM, 2011.

[10] S. M. Mniszewski, C. Junghans, A. F. Voter, D. Perez, and S. J. Eidenbenz. Tadsim: Discrete event-based performance prediction for temperature-accelerated dynamics. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 25(3):15, 2015.

[11] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. A detailed gpu cache model based on reuse distance theory. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 37–48. IEEE, 2014.

[12] N. Santhi, S. Eidenbenz, and J. Liu. The simian concept: Parallel discrete event simulation with interpreted languages and just-in-time compilation. In *Proceedings of rate 2015 Winter Simulation Conference*, page to appear, 2015.

[13] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *ACM SIGPLAN Notices*, volume 47, pages 11–22. ACM, 2012.

[14] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[15] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2sim: a simulation framework for cpu-gpu computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 335–344, 2012.

[16] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[17] D. Wade. Computational science graduate fellowship program. Technical report, NNSA, 07 2015.

[18] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010.