Towards Predicting Performance of GPU-dependent Applications on the Example of Machine Learning in Enterprise Applications

Felix Willnecker
willnecker@fortiss.org
fortiss GmbH, Munich, Germany

Helmut Krcmar
krcmar@in.tum.de
Technical University of Munich, Garching, Germany

Abstract

Algorithms processed by Graphics Processing Units (GPU) became popular recently. Bitcoin mining algorithms, image processing and all types of machine learning are famous examples for that. Infrastructure-as-a-Service provider picked up this trend and offer graphics processing power as part of their service portfolio. The performance gains when choosing a GPU implementation can be enormous. Designing and implementing a GPU-depended algorithm has some fundamental differences compared to classical algorithms, but not all algorithmic problems benefit from GPU usage regarding the overall performance and response time. Especially the interaction between Central Processing Unit (CPU) and GPU must be considered as it can become a bottleneck. Predicting and comparing the performance of GPU-depended applications in combination with their corresponding CPUs allows to assist design decisions in modern applications. In this work, we present concepts on how to predict algorithm performance relying on GPU processing and their relationship with the CPU using the Palladio Component Model and the Palladio Bench.

1 Introduction

Applications strongly utilizing Graphics Processing Units (GPUs) have for long been in the domain of Scientific Computing, High Performance Computing (HPC), or Gaming [2]. Nowadays, highly parallel algorithms become more important in other domains such as Enterprise Applications (EAs) due the adoption of new algorithms and the usage of cheap and fast GPUs [4, 7]. Several application domains such as machine learning, image recognition, the famous Bitcoin mining and industry use-cases such as predictive maintenance make use of highly parallel algorithms. However, current Architecture-level performance models are not prepared to handle these kinds of applications.

Predicting the performance of EAs using Architecture-level performance models such as Palladio Component Model (PCM) proved to have a lot of benefits, especially in combination with automatic model generation [1, 9, 12, 13]. Modern applications, especially those conducting machine learning, rely both on Central Processing Unit (CPU) and GPU. CPU usually handles the control flow and GPU processes multiple small calculations in parallel. Prediction capabilities for such applications are necessary in order to apply use-cases such as capacity planning, architecture optimization, and model-based performance analysis to these types of applications [3, 5]. This work presents approaches to integrate GPU prediction capabilities into PCM. This would allow to model, predict and optimize the performance of modern applications that use algorithms such as GPU-optimized machine learning.

2 Related Work

Resios et al. (2011) present an approach to model and predict GPU performance based on NVIDIAs CUDA Parallel Computing Platform[1] [5]. They focus on determining the potential computing power in order to predict optimization potential [5]. Their parametrized model is custom developed and evaluated using matrix operations. They outline, that especially the relationship with the potential computing power and the memory and its bandwidth need to be considered.

Kothapalli et al. (2009) predict GPU performance with a strong focus on memory access strategies [2]. Furthermore, the overhead of transferring data between GPU and CPU need to be considered, so that an integrated model containing both resource types becomes necessary.

Boyer et al. (2013) demonstrate a model that predicts the performance when porting an application from CPU to GPU [6]. Their findings suggest that especially data intensive applications might not benefit from a migration to the GPU as the transfer overhead erases potential performance gains. A holistic model must consider data and data transfer in order to predict performance metrics of GPU-dependent applications reliable.

Frank and Hilbrich (2016) showed that multicore CPU prediction in PCM are not fully supported yet [11]. They executed a series of experiments and compared them with simulations. Although, the results are promising some weaknesses were uncovered. Missing support for memory transfer delay in PCM are

[1]`http://www.nvidia.com/object/cuda_home_new.html`

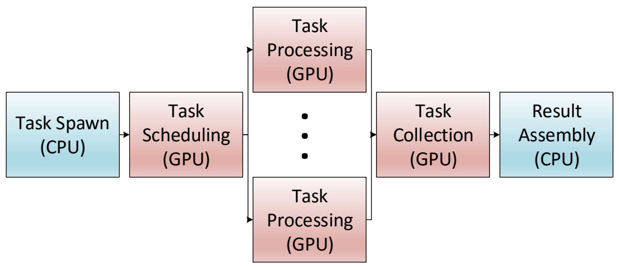

Figure 1: Basic task model and interaction between CPU and GPU

suspected as one of the main reasons for that.

3 Approach

GPUs are designed to calculate small operations or programs in parallel. Modern consumer graphic cards support between 1600 and 2560 parallel operations at relatively low clock speed between 1 and 1,6 Gigahertz (GHz). Professional graphic cards support 5000 and more parallel operations. Infrastructure as a Service (IaaS) providers like Amazon Web Services (AWS) already offer such systems with up to 16 GPUs[2]. Therefore, 80,000 parallel operations are theoretically possible in such a scenario.

In contrast classical CPUs only support 8 - 16 parallel operations for consumer CPUs and a few hundred operations for professional server CPUs. This massive parallelization provides some challenges for PCM as it was designed for single or few core operations.

One major challenge is the interaction between CPU and GPU. Depending on the underlying framework (e.g., OpenCL, DirectCompute, CUDA, etcs.) the task model and execution as well as the interaction between GPU and CPU might be different. However, in general CPU controls the application and its control flow and assigns tasks to GPU. The GPU then executes these tasks in parallel and returns the results to CPU. Figure 1 presents a simplified task model, which we try to simulate using PCM.

We simulate a simple GPU task batch representing processing of a neuronal network (Multilayer Perceptron), using PCM standard meta-model and SimuCom as depicted in Figure 2 [1, 12]. Our initial evaluation is based on the PCM minimum example project and added a *Fork Action* to the only *SEFFF* of this example. First, we generated 1,000 threads in this action with simple operations. Furthermore, we used the CPU resource to simulate GPU operations. Therefore, we added a slow *Processing Rate* and a large number of replicas. A 10 minutes timeframe simulation for a single user took over 5 min of computation time on an Intel I7-3520M running at 2.90 GHz. This would be sufficient for simple con-

[2]https://aws.amazon.com/de/blogs/aws/new-p2-instance-type-for-amazon-ec2-up-to-16-gpus/

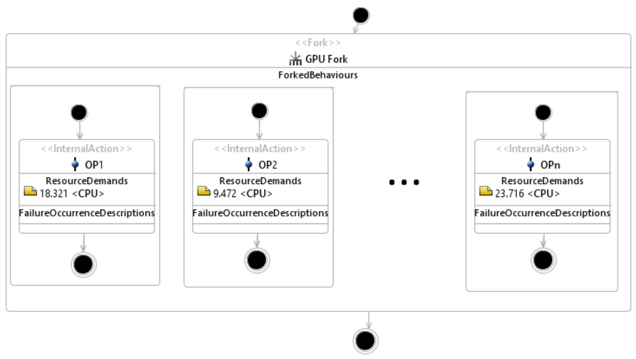

Figure 2: Example PCM model for typical GPU operations

sumer GPU based operations using a single resource container with a single GPU. Second, we scaled up to 5,000 threads to simulate a professional GPU. The generation step takes more than 12 minutes and the generated code was not executable as the Java class limit of 65,535 bytes[3] exceeded. Executing the same model using EventSim leads to a Stackoverflow caused by a large number of parallel threads.

In order to simulate such models, we need to simplify the processing of parallel executions. This work suggests two changes to reduce simulation and model complexity as well as extend the functionality of PCM:

- Simplify the Fork Behavior Action
- Add new GPU resource and memory specification

ForkedBehaviors allow to model complex architectures within the fork, which causes code and execution overhead for the simulation. In order to fit for the use cases of GPU processing, we suggest the definition *RDSEFFs* instead of *ForkedBehaviors* as children of each *Fork*. This allows the seperation of the generated simulation code into many different files. Furthermore, we could parameterize the *SEFF* calls and reduce these calls in our case to a simple list of *ResourceDemands*. Furthermore, *RDSEFF* re-usage would reduce the size of the generated classes dramatically while still allowing to use the full functionality of PCM when simulating threads.

ForkedBehaviors would then act similar to *ExternalCalls* and also reflect that the thread internal behavior is usually implemented in dedicated components, while thread control is conducted by a dedicated controlling component [8]. This extension would reduce the complexity of the simulation and ease the model creation/generation. However, it disregards the relationship between CPU and GPU and it ignores memory and its bandwidth as a potential performance bottleneck.

To solve these problems, we suggest to add a dedicated *GraphicalProcessingUnitSpecification* represent-

[3]https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html

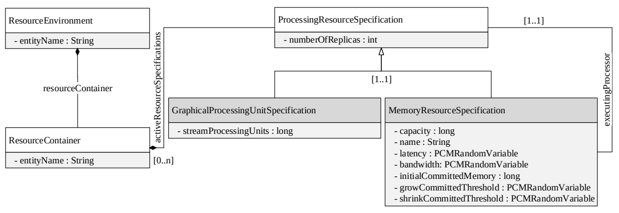
ResourceEnvironment
- entityName : String
resourceContainer
ResourceContainer
- entityName : String
activeResourceSpecifications
[0..n]
ProcessingResourceSpecification
- numberOfReplicas : int
[1..1]
[1..1]
GraphicalProcessingUnitSpecification
- streamProcessingUnits : long
MemoryResourceSpecification
- capacity : long
- name : String
- latency : PCMRandomVariable
- bandwidth: PCMRandomVariable
- initialCommittedMemory : long
- growCommittedThreshold : PCMRandomVariable
- shrinkCommittedThreshold : PCMRandomVariable
executingProcessor

Figure 3: GPU and memory extension for PCM

ing one GPU which extends *ProcessingResourceSpecification* by a parameter describing its stream processing units. Therefore, we do not need to misuse the *numberOfReplicas* attribute and can define the GPU as is. Adding a dedicated resource allows to distinguish between resource demanding calls to CPU and GPU.

Furthermore, we suggest to extend the *MemoryResourceSpecification* introduced in our previous work to represent VRAM of a GPU [10]. The previous implementation of this resource allocated and retrieved memory instantly. This disregards the effects of memory bandwidth and latency [2]. Similar to the *LinkingResource*, we suggest to add bandwidth and latency to this resource.

Both extensions are depicted in Figure 3. They would allow to simulate typical GPU uses cases like machine learning, image processing or predictive maintenance and thus extend PCM for next generation EA.

4 Conclusion

This work presents current deficits in PCM and its simulation engine that prevents simulating massive parallel algorithms. Typical use cases that are strongly GPU-dependent emerged in new EA and require capacity planning and optimization. We presented two extension that allow to model and simulate GPU the relation to the CPU and regards the potential bottleneck memory. The extensions require minor changes to the meta-model and minor changes to the simulation engine.

References

[1] S. Becker, H. Koziolek, and R. Reussner. “The Palladio Component Model for Model-Driven Performance Prediction”. In: *Journal of Systems and Software (JSS)* 82.1 (2009). Special Issue: Software Performance - Modeling and Analysis, pp. 3–22.

[2] K. Kothapalli et al. “A performance prediction model for the CUDA GPGPU platform”. In: *2009 International Conference on High Performance Computing (HiPC)*. Dec. 2009, pp. 463–472.

[3] A. Koziolek, H. Koziolek, and R. Reussner. “PerOpteryx: Automated Application of Tactics in Multi-objective Software Architecture Optimization”. In: *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*. ACM. 2011, pp. 33–42.

[4] J. Krueger et al. “Applicability of GPU Computing for Efficient Merge in In-Memory Databases.” In: *ADMS@ VLDB*. 2011, pp. 19–26.

[5] A. Resios and V. Holdermans. “GPU performance prediction using parametrized models”. In: *Master’s thesis, Utrecht University, The Netherlands* (2011).

[6] M. Boyer, J. Meng, and K. Kumaran. “Improving GPU Performance Prediction with Data Transfer Modeling”. In: *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. May 2013, pp. 1097–1106.

[7] S. Grauer-Gray et al. “Accelerating Financial Applications on the GPU”. In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. GPGPU-6. Houston, Texas, USA: ACM, 2013, pp. 127–136.

[8] D. C. Schmidt et al. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. Vol. 2. John Wiley & Sons, 2013.

[9] A. Brunnert and H. Krcmar. “Continuous Performance Evaluation and Capacity Planning Using Resource Profiles for Enterprise Applications”. In: *Journal of Systems and Software (JSS)* (2015).

[10] F. Willnecker et al. “Full-Stack Performance Model Evaluation using Probabilistic Garbace Collection Simulation”. In: *Proceedings of the 2015 Symposium on Software Performance (SSP 2015)*. 2015.

[11] M. Frank and M. Hilbrich. “Performance Prediction for Multicore Environments—An Experiment Report”. In: *Softwaretechnik-Trends* 36.4 (2016).

[12] R. H. Reussner et al. *Modeling and Simulating Software Architectures: The Palladio Approach*. MIT Press, Oct. 2016.

[13] F. Willnecker and H. Krcmar. “Optimization of Deployment Topologies for Distributed Enterprise Applications”. In: *Proceedings of the 2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA 2016)*. 2016.