A Queueing Model of Hybrid Parallel Pipelines PDPTA'15

Fahad Khalid*, Lena Herscheid, Andreas Polze
Hasso Plattner Institute for Software Systems Engineering
14482 Potsdam, Germany
fahad.khalid, lena.herscheid, andreas.polze@hpi.uni-potsdam.de

Abstract— *Hybrid parallel pipelines are suited for many scientific algorithms, which can be sped up significantly by using CPU and GPU hardware. However, the complexity of modern applications and hardware makes it hard to optimally configure hybrid parallel pipelines. To this end, analytical modeling approaches are needed.*

We present a novel Queuing Theory model of hybrid parallel pipelines, which can be used for performance prediction and optimization. Based on this model, we propose an algorithm for automatically identifying the bottleneck stage and tuning its degree of parallelism. Empirical evidence from various pipeline configurations demonstrates the validity of our model.

Keywords: parallel pipeline, optimization, hybrid architectures, queueing theory, queueing model

1. Introduction

Emerging hybrid parallel architectures (including both CPUs and accelerators such as *graphics processing unit*s (GPUs)) make huge performance gains possible, if the hardware is used efficiently and idle times are avoided. However, the problem of partitioning work cleverly over different processors while also keeping data access and transfer latencies low, is a hard one to solve.

In this context, *pipeline parallelism* is a pattern suited to many scientific applications. The computation on data items is performed in a sequence of pipeline *stages*. Each pipeline stage can be parallelized internally, by exploiting data or task parallelism. Since the different stages can operate on different items independently of each other, computation can be overlapped, leading to performance gains.

Within software pipelines, the question arises of how compute resources should be distributed across pipeline stages, and how the issues of load imbalance and data transfer bottlenecks should be addressed. The tuning of such parameters is a complex task depending on various system parameters. To this end, analytical models enable predicting the performance of different pipeline configurations.

Queueing Theory [1] is a mathematical modeling tool for processes, where pending items in *queues* are processed by *servers*, whose throughput is expressed as stochastic distributions. A *queueing network* consists of different servers, linked together by their input and output queues. Queueing networks allow for the computation of the overall throughput, utilization metrics and the expected number of waiting items. They are therefore a powerful tool also for predicting the performance of parallel pipelines. We present a Queueing Theory model of hybrid parallel pipelines and use it to guide performance optimizations. In particular, we show how GPU stages can be represented realistically in such a queueing network. Our model can be employed to tune the assignment of threads to pipeline stages.

2. Related Work and Our Contribution

Foremost, our work builds upon that of Navarro et al. [2], [3]. In their research, a queueing network model is presented for multithreaded CPU-only applications. This queueing model, which consists of an exponentially distributed queue for each pipeline stage, is then used to optimize the number of tokens (i.e., work items) in the system and the number of threads assigned to each pipeline stage.

Analytical Performance Models for software pipelines: Gonzalez et al. [4] propose an analytical model for software pipelines which can be used to optimally assign pipeline stages to a ring of processors.

Wei et al. [5] analytically minimize the communication overhead in accelerator-based architectures using *integer linear programming* (ILP). This minimization problem has also been tackled by model checking data flow graphs [6].

Analytical Performance Models for GPUs: The performance of hybrid parallel architectures is hard to predict, as it depends on multiple platform and application dependent parameters. There have been multiple attempts at analytically characterizing the performance of such hybrid architectures.

The *GPURoofline* model [7], as a GPU-focussed extension of the roofline model [8], enables the prediction of a theoretical upper bound on kernel performance, and the evaluation of optimizations, which improve the communication to computation ratio. The *boathull* model [9] builds upon GPURoofline, additionally including host to accelerator data transfer. Algorithms are classified by the number of parallel work units, the amount of computation,

data I/O, and memory access patterns. Prior to the actual development of kernel source code, the boathull model can be used to make performance predictions.

GROPHECY [10] is a framework for predicting GPU performance by means of CPU code skeletons. Several parameters regarding the code layout (the number of distinct code blocks, and tasks assigned to each thread) and regarding the workload (the number of memory accesses and instructions per thread), are used to project GPU performance without having to write actual GPU code.

Low-level models, which are based on the analysis of kernel source code or even GPU assembly, have also been discussed [11], [12], [13].

Optimizations for Software Pipelining: Pipeline parallelism has been identified as a commonly occurring pattern [14], lending itself to performance optimizations by overlapping different pipeline stages. Software Pipelines exhibit a special distribution of workload, and are characterized with special benchmarks such as `ferret` and `dedup` [15]. Various approaches to auto-tuning software pipelines have been discussed recently.

On-the-fly pipeline parallelism [16] determines the pipeline structure dynamically during runtime. Integrated into a Cilk-based work stealing system, this approach has demonstrated good performance in pipeline benchmarks.

Feedback-directed pipeline parallelism [17] is the auto-tuning of core-to-stage allocation by repeatedly finding the slowest pipeline stage and gradually providing it with more resources, until no more performance is gained. This hill climbing approach, has been shown to improve performance in CPU-only software. It can be integrated into software libraries such as TBB.

Load-balanced pipeline parallelism [18] tries to extract fine-grained pipeline parallelism from loops by analyzing the program graph. Different types of intra- and cross-iteration dependencies are identified and barriers for serial stages are inserted automatically. The approach is implemented in a compiler and does not yet tackle GPU architectures.

Parallel-Stage Decoupled Software Pipelining (PS-DSP) [19], an extension of *Decoupled Software Pipelining* (DSWP) [20], is the extraction of pipeline parallelism from loops in the presence of loop-carried dependencies, for compiler-based automatic code transformations. A more recent extension to this work is *speculative DSP* [21], which speculatively ignores infrequent dependencies.

Van Der Wijngaart et al. [22] have shown how fine grained software pipelines (at the instruction level) can be modeled and optimized analytically.

Stream Programming Models: Another vast body of related work studies how streams of data can be modified efficiently using a sequence of compute kernels. The focus of stream programming has recently shifted from multimedia applications to other data-intensive computations. It has been acknowledged that the stream programming style lends itself to pipeline parallel optimizations [23]. A catalogue of stream processing optimizations is presented in [24].

Thies et al. [25] presented an annotation-based tool for C programs, which facilitates the representation of coarse grained software pipelines. Based on Valgrind and a custom runtime for the resulting pipeline application, this approach has proven helpful for various streaming applications.

There have been multiple approaches to mapping streaming programs, often written in a dedicated language, to multicore architectures [26], [27]. An extension of the *OpenMP* standard, which incorporates language constructs for expressing streams and pipeline parallelism, has been proposed [28]. Recently, GPUs have also become a target platform for the execution of streaming applications [29].

2.1 Research Gap and Our Contribution

Past research has been done on predictive analytical models, as well as on optimization approaches for certain architectures, often using runtime information. Our contribution lies in the intersection of these two research areas. We first present an analytical Queueing Theory model of pipeline parallelism, which can predict the throughput of different pipeline configurations. On the basis of this model, we also demonstrate how to optimize away the bottleneck stage of a pipeline automatically. The predictions from our model can be used to avoid the time-consuming effort of manually trying out numerous pipeline configurations.

In contrast to previous research, in particular that of Navarro et al. [2], our model describes hybrid GPU-CPU systems, where pipeline stages can run either on accelerator hardware, or on the host. Thus, we contribute a novel way of modeling hybrid parallel pipelines both for performance prediction, and for optimization.

3. The Concept of Hybrid Pipeline

We define a hybrid parallel pipeline as an implementation of the parallel pipeline pattern in such a way that the at least one pipeline stage is executed on a conventional CPU, and at least one other pipeline stage is executed on an accelerator architecture. In this paper, we assume the accelerator architecture to be a Graphics Processing Unit (GPU). Throughout the rest of the paper, we will refer to the CPU as *Host*, and the GPU as *Device*.

We further assume that *Device* memory is separate from the *Host* memory. Therefore, in order for the *Device* to process data, the data must first be transferred from the *Host* to the *Device*, and finally the results transferred from the *Device* to the *Host*.

The total computation time can be reduced by overlapping kernel execution and data transfer. This is achieved by

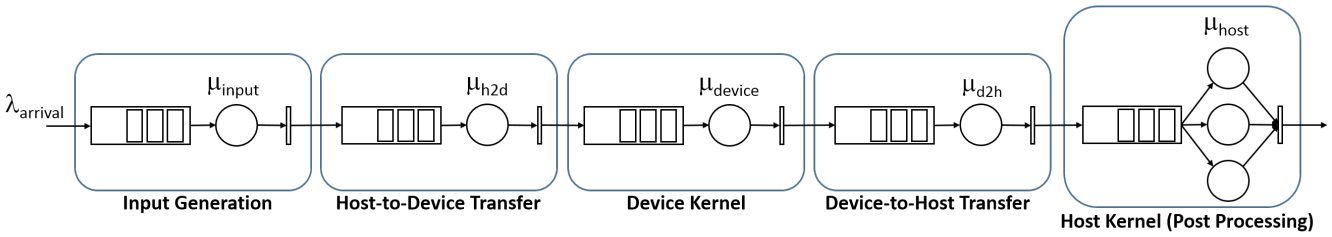

Fig. 1: A typical hybrid pipeline. Stages include data transfer to and from the *Device*, a computation kernel on the *Device*, and a Post Processing and Input stage on the *Host*. Each stage can be modeled as an $M/M/c$ queue, where $c = 1$ for *Device* and serial *Host* stages. The service rates of the different stages μ_i determine the overall throughput and bottleneck stage.

using asynchronous (non-blocking) routines for data transfer between *Host* and *Device*.

Figure 1 depicts a typical hybrid parallel pipeline. Following is a description of the stages:

1) *Input*: The purpose of this stage is to generate input data that is processed by the following stages. This stage can either comprise reading input files, or pre-processing already read data. Here we assume that this stage is executed on the *Host*, and that it is possible to utilize more than one *Device* threads to process this stage.
2) *H2D*: This stage represents the asynchronous transfer of input data from *Host* to *Device*. We assume that this stage can only be processed by one *Host* thread, since there is only one channel available for transfer of data from *Host* to *Device*.
3) *Device Kernel*: This stage represents the *Device* kernel. We assume that the *Device* cannot process more than one kernel at a time, and therefore only requires one *Host* thread. In this case, parallelism is within the *Device* kernel; multiple *Device* kernels cannot be executed in parallel.
4) *D2H*: This stage represents the asynchronous transfer of result data from *Device* to *Host*. We assume that this stage can only be processed by one *Host* thread, since there is only one channel available for transfer of data from *Device* to *Host*.
5) *Post Processing*: This stage represents a *Host* kernel that is used to post-process results generated by the *Device*. We assume that it is possible to utilize multiple *Host* threads to process this stage.

3.1 Item flow through the pipeline

Here we use the above mentioned description of pipeline stages to present a hypothetical scenario for the flow of items through the pipeline.

1) The *Input* stage generates multiple items; one item is generated per *Host* thread used for this stage.
2) Items generated by the *Input* stage are queued in the *H2D* stage. This stage transfers input data from *Host* to *Device* for one item at a time in a *first come first served* manner.
3) Items forwarded by the *H2D* stage are queued in the *Device Kernel* stage. This stage executes the *Device* kernel for one item at a time in a *first come first served* manner.
4) Output items generated by the *Device Kernel* stage are queued in the *D2H* stage. This stage copies output data from *Device* to *Host* for one item at a time in a *first come first served* manner.
5) The *Post Processing* stage can use multiple threads to process multiple output items in parallel, queued by the *D2H* stage.

4. Analytical Model of Hybrid Parallel Pipelines

We use Queuing Theory to model a hybrid parallel pipeline as a network of queues. In this Section, we first define the relevant parameters of the queuing model, and then map these to the hybrid parallel pipeline example discussed earlier.

The following parameters are used for each queue in the model.

- $T_{arrival}$ = inter-arrival time, i.e., time duration between the arrival of two successive items in the queue
- $T_{service}$ = service time, i.e., time it takes for the service to process one item
- c = number of servers

The above mentioned definitions can be used to derive the following quantities:

- *Arrival rate*, defined as item arrival per unit time,

$$\lambda = \frac{1}{T_{arrival}} \tag{1}$$

- *Service rate*, defined as items serviced per unit time,

$$\mu = \frac{1}{T_{service}} \tag{2}$$

For each queue, we assume exponential distributions for both $T_{arrival}$ and $T_{service}$. This assumption implies that the service and arrival rates are, at some point, in a steady state

and do not vary much. Such a queue can be represented as a $M/M/c$ queue in Kendall's notation. Each stage in the pipeline of Figure 1 is a $M/M/c$ queue. For *H2D*, *Device Kernel*, and *D2H* stages, we assume $c = 1$. For *Input* and *Post Processing* stages, $c \geq 1$. The complete pipeline is modeled as a network of $M/M/c$ queues.

Let λ_i be the arrival rate for stage i, and μ_i be the service rate for stage i. Then,

$$\lambda_i = \mu_{i-1} \tag{3}$$

i.e., arrival rate at stage i is equal to the service rate at stage $i-1$.

We define throughput of the stage as,

$$\frac{number\ of\ processed\ items}{time} = service\ rate = \mu \tag{4}$$

Then, given that service rate for the slowest stage defines the upper bound on the overall pipeline throughput, we define the overall pipeline throughput as,

$$X = min\,(M) \tag{5}$$

where, $M = \{\mu_s, \forall s \in S\}$, and S is the set of all stages.

In order to compare the predictions of the model with experiment results, we define a quantity that is derived from the above mentioned parameters, but can also be measured directly. This metric is time, i.e., the total time taken by the pipeline. Using Little's Law, it is defined as,

$$T = \frac{n}{X} \tag{6}$$

where n is the total number of items processed by the pipeline.

4.1 Optimizing Pipeline Parameters

In the above mentioned model, arrival times and service times are measured using serial execution of the pipeline stages. Then, a degree of freedom exposed in the model that can be used to optimize total throughout, is the number of threads per parallel *Host* stage, i.e., c.

As mentioned earlier, the total throughput of a pipeline is constrained by the throughput of the slowest stage. Therefore, we can optimize total throughput using c if and only if the slowest stage is a *Host* processing stage which can be modeled as a queue with multiple servers. *Input* and *Post Processing* stages of Figure 1 fit this criterion.

Let us call the slowest stage in the pipeline the *bottleneck stage*. For simplicity, we assume that only one of the stages can be the bottleneck stage. Then, we can identify the bottleneck stage as,

$$S_{bottleneck} = \{S_i \in S | \mu_{s_i} = min\,(M)\} \tag{7}$$

Now, let us assume that $S_{bottleneck}$ is a *Host* processing stage that satisfies the condition: $c \geq 1$. Then let us define a new parameter called *traffic intensity* as,

$$\rho = \frac{\lambda}{c\mu} \tag{8}$$

Using the *stability condition* for a $M/M/c$ queue, we have,

$$\frac{\lambda}{c\mu} < 1 \tag{9}$$

This implies that if $c = 1$, and $\frac{\lambda}{\mu} > 1$, the optimal number of threads for $S_{bottleneck}$,

$$c_{bottleneck} \geq \frac{\lambda}{\mu} \tag{10}$$

This provides us with the lower bound on $c_{bottleneck}$. In order to compute the optimal value, we can simply measure the value for c such that,

$$\frac{\lambda}{\mu} \leq c \leq c_{max} \tag{11}$$

where, c_{max} is the maximum number of threads available for $S_{bottleneck}$.

The number of *live items*, n_{live}, is another variable that affects pipeline throughout. It can be defined as the number of items that can be processed simultaneously over the entire pipeline. In our model, we assume $n_{live} = max\{c, N\}$, where N is total number of pipeline stages.

Algorithm 1 summarizes the process of finding the optimal number of servers per stage.

Algorithm 1: Algorithm for finding the optimal number of threads per server to maximize pipeline throughput. $c_{min\ \mu}$ is the c corresponding to the lowest μ determined within the foreach loop.

Input : P: Set of parallel stages
M: Throughputs of all pipeline stages
Output: Optimal value of c for bottleneck stage

```
S_bottleneck = {S_i ∈ S | μ_{s_i} = min(M)};
if S_bottleneck ∈ P then
    c_start = λ/μ;
    foreach c: from c_start to c_max do
        Store μ corresponding to c;
    end
    c_optimal = c_{min μ};
end
```

5. Empirical Analysis

In this Section, we use two different hybrid parallel pipelines to evaluate the optimization algorithm presented earlier. We begin by providing essential information about the test environment, and proceed with descriptions of the two use cases. We compare the results of executing the pipelines with the results predicted by the queuing model.

5.1 Test Environment

The empirical measurements discussed in this paper were conducted on an Intel Nehalem EX architecture based quad-core Xeon E5520 CPU with 4 cores, each of which supports 2 hardware threads. The machine contains 17GB of RAM. The installed operating system is Ubuntu 12.04 LTS. As an additional device, the machine contains an NVIDIA GTX 680 GPU with 4GB of device memory, which supports compute capability 3.0. All tests were run using CUDA driver version 5.5. Our tested code can also be used with CUDA 5.0, which is the earliest version supporting callback functionality. The compilers we used are GCC version 4.6.3 and NVCC version 5.5.

5.2 Use Case I – Toy Pipeline

As the first use case, we use a $5-stage$ toy pipeline. The purpose of this use case is to be able to experiment with different pipeline configurations, in order to thoroughly test the queuing model. Real life examples tend to be much less flexible, since the purpose of such codes is to solve specific problems. Our toy pipeline framework, however, makes it possible for us to test the following different scenarios: 1) *Post Processing* constitutes the bottleneck stage, 2) *Input* constitutes the bottleneck stage, and 3) *Device Kernel* eventually becomes the bottleneck stage.

Stages in the toy pipeline are based on the concepts described in Section 3. The implementation allows us to run the pipeline with different λ and μ values, configurable for each stage. It is also possible to configure c for the *Input* and *Post Proccessing* stages. The pipeline has been implemented using the Hybrid Pipeline Framework (HyPi) [30].

5.2.1 Results

Table 1 presents results for three test scenarios. The topmost entry shows a setup where the *Post Proccessing* stage comprises the pipeline bottleneck. We see that the queuing model correctly identifies the bottleneck stage. We further observe that the optimization algorithm correctly identifies the optimal value of c for the bottleneck stage. Similar results are presented in the second table entry, a scenario where the *Input* stage constitutes the bottleneck.

Table 1: Comparison of $S_{bottleneck}$ and $c_{optimal}$ values predicted by the model, with values measured from the simulation runs. Each row indicates one of the scenarios mentioned in Section 5.2.1. The predicted values are computed as floating point values, which are rounded up to the nearest integer.

$S_{bottleneck}$		$c_{optimal}$	
Predicted	Measured	Predicted	Measured
PP	PP	$\lceil 1.43 \rceil = 2$	2
Input	Input	$\lceil 1.35 \rceil = 2$	2
PP, Input	PP, Input	$\lceil 1.35 \rceil$, $\lceil 3.78 \rceil$	2, 4

The last entry in Table 1 presents results for a scenario where we first optimize c for the *Post Proccessing* stage, which shifts the bottleneck to the *Input* stage. We then optimize c for the *Input* stage, which results in the *Device Kernel* forming the final bottleneck stage. This scenario runs the optimization algorithm multiple times, until there is no further opportunity for optimization.

5.3 Use Case II – Combinatorial Candidate Generation

In this Section, we present results of applying the queuing model to a real simulation from the domain of Computational Biology. *Combinatorial Candidate Generation* is the most compute intensive part of the Nullspace [31] algorithm for enumerating elementary flux modes in metabolic networks. It has been shown in previous work [32] that using a hybrid parallel pipeline is an effective strategy to improve the performance of *Combinatorial Candidate Generation*.

A metabolic network comprises *metabolites* – chemical compounds – and *reactions*. A path in the network consists of one or more *substrate* metabolites being converted into one or more *product* metabolites. Such a system can be modeled as a node-weighted directed hypergraph, where nodes represent metabolites and edges represent reactions. The Nullspace [31] algorithm is used to enumerate all Elementary Flux Modes (EFMs) in the network, where an EFM is a minimal subnetwork that operates at equilibrium. For the sake of brevity, we refer the reader to [33] for a detailed description of the Nullspace algorithm. Here, we only present a brief description of how the combinatorial candidate generation algorithm is mapped on to a hybrid parallel pipeline.

Following is a description of the $3-stage$ hybrid parallel pipeline used for the combinatorial candidate generation algorithm:

1) *Generate*: This stage is implemented as a *Device* kernel. The kernel implements a combinatorial algorithm, which takes as input two bit-matrices. A bitwise-OR operation is performed on all possible combinations of columns of the two matrices. On each vector resulting from the bitwise-OR operation, a *popcount* operation is performed. If the *popcount* value is greater than a certain predefined threshold τ, a result bit corresponding to the combination vector is set in the result vector.
2) *D2H*: Due to the combinatorial nature of the *Device* kernel, the result vector can be very large, even for small input matrices. This stage is responsible for the asynchronous transfer of the result vector from *Device* to *Host*.
3) *Map*: This stage is executed on the *Host*. It parses the result bit-vector generated by the *Device* kernel, and maps all set bits to the respective column indices. This is a memory-intensive operation with low arithmetic intensity, and is therefore implemented as a *Host* stage.

5.3.1 Results

As mentioned in Section 3, the *Device Kernel* and *D2H* stages are implemented as serial stages due to the inherently serial nature of the stages. This holds true for the *Generate* and *D2H* stages in this pipeline. Therefore, *Map* is the only stage with the possibility to optimize the number of parallel threads, c.

Results of the simulation with different c values are plotted in Figure 2. We observe that the queuing model accurately predicts the pipeline performance. Also, $c_{optimal}$ calculated using Algorithm 1 is in agreement with the simulation results.

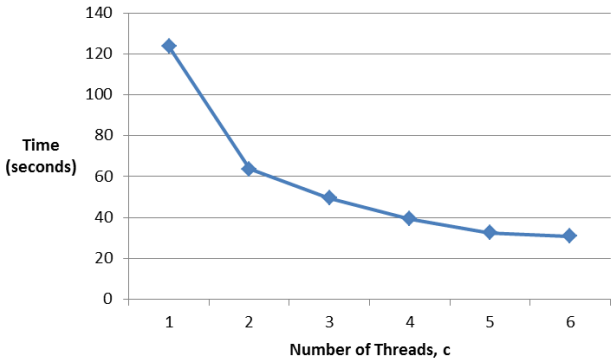

Fig. 2: Measured times for the Combinatorial Candidate Generation pipeline against different values of c. The model predicted $c = 5.37$ for the *Map* stage, which we round up to 6. Simulation runs confirm that $c = 6$ results in optimal performance.

A particularly interesting aspect of this simulation is the fact that the *Generate* stage automatically partitions the result vector according to the available *Device* memory. This makes it possible to run simulations for arbitrarily large datasets, since partitions are processed one at a time. A computed partition is transferred from *Device* to *Host*, which leads to the processing of the next partition. The maximum partition size is always kept lower than the available *Device* memory size.

Common wisdom would lead us to believe that partition size should be kept as large as possible in order to improve the *Device* to *Host* data transfer. Figure 3 shows the performance impact of different partition sizes on simulation performance. It can be observed that the performance degrades only for a very large number of partitions. Up to a certain small number of partitions, the performance does not vary significantly with the varying number of partitions.

The above mentioned result is very important for pipeline performance. If we use a single large partition, it can only be consumed by one *Map* server. In order to exploit $c > 1$ in the *Map* stage, it is essential that we generate multiple smaller partitions so that these can be consumed in parallel.

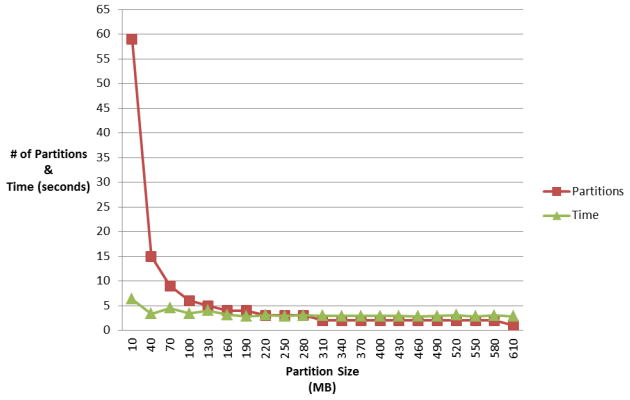

Fig. 3: Correlation between number of partitions and total simulation time. The $x-$axis depicts increasing partitions sizes. As partitions size increases, the number of partitions decreases. We observe that number of partitions only affects performance significantly for a high number of partitions. Afterwards, the performance stabilizes.

Note that this holds true only if the following stage supports $c > 1$. In the toy pipeline presented in Section 5.2, the *Input* stage would not benefit from this strategy, since the stage that follows is inherently serial.

6. Conclusion

Summary: We have presented an approach to mathematically model hybrid parallel pipelines using Queuing Theory. The model is complemented by an algorithm that can be used to identify the bottleneck stage, as well as the optimal number of threads to use for *Host* processing stages that can process items in parallel. We have provided empirical evidence to support our claims.

Discussion: Throughout the paper, we have assumed that a *Device Kernel* stage is executed on a single *Device* such as GPU. Here we propose that the model can be extended to describe multi-GPU systems as well. This would require *H2D*, *D2H*, and *Device Kernel* stages with $c =$ number of GPUs.

The model is also designed to assume a one-to-one mapping between software and hardware threads. This is due to the fact that the performance of an oversubscribed system is dependent on scheduling algorithms used by the pipeline framework and the operating system. This means that it is not possible to present results for such systems without loss of generality. Therefore, we decided not to include results of modeling oversubscribed systems.

In the future, we would like to investigate the possibility of utilizing further insights from Queuing Theory to improve the design and performance of hybrid parallel pipelines. Moreover, it would be interesting to extend the concept of

Device to other types of accelerators, such as the Intel Xeon Phi co-processor. At this point, we do not know whether the model presented in this paper can be extended to other *Devices*.

References

[1] R. Jain, "The art of computer system performance analysis: techniques for experimental design, measurement, simulation and modeling," *New York: John Willey*, 1991.

[2] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '09, 2009, pp. 281–290.

[3] A. Navarro, R. Asenjo, S. Tabik, and C. Çaşcaval, "Load balancing using work-stealing for pipeline parallelism in emerging applications," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09, 2009, pp. 517–518.

[4] D. González, F. Almeida, L. Moreno, and C. Rodríguez, "Towards the automatic optimal mapping of pipeline algorithms," *Parallel Comput.*, vol. 29, no. 2, pp. 241–254, Feb. 2003.

[5] H. Wei, J. Yu, H. Yu, and G. R. Gao, "Minimizing communication in rate-optimal software pipelining for stream programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10, 2010, pp. 210–217.

[6] A. Malik and D. Gregg, "Orchestrating stream graphs using model checking," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 3, pp. 19:1–19:25, Sept. 2008.

[7] H. Jia, Y. Zhang, G. Long, J. Xu, S. Yan, and Y. Li, "GPURoofline: A model for guiding performance optimizations on GPUs," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12, 2012, pp. 920–932.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[9] C. Nugteren and H. Corporaal, "The boat hull model: Enabling performance prediction for parallel computing prior to code development," in *Proceedings of the 9th Conference on Computing Frontiers*, ser. CF '12, 2012, pp. 203–212.

[10] 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*, ser. SC '11, 2011, pp. 14:1–14:11.

[11] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11, 2011, pp. 382–393.

[12] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 152–163, June 2009.

[13] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," *SIGPLAN Not.*, vol. 45, no. 5, pp. 105–114, Jan. 2010.

[14] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, Eds., *Sourcebook of Parallel Computing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[15] in *Computer Architecture*, ser. Lecture Notes in Computer Science, A. Varbanescu, A. Molnos, and R. van Nieuwpoort, Eds., 2012, vol. 6161.

[16] I. Lee, T. Angelina, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang, "On-the-fly pipeline parallelism," in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2013, pp. 140–151.

[17] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-directed pipeline parallelism," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, 2010, pp. 147–156.

[18] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Load-balanced pipeline parallelism," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, 2013, pp. 14:1–14:12.

[19] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, "Parallel-stage decoupled software pipelining," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08, 2008, pp. 114–123.

[20] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04, 2004, pp. 177–188.

[21] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, 2007, pp. 49–59.

[22] R. F. Van der Wijngaart, S. R. Sarukkai, and P. Mehra, "Analysis and optimization of software pipeline performance on MIMD parallel computers," *Journal of Parallel and Distributed Computing*, vol. 38, no. 1, pp. 37–50, 1996.

[23] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 151–162, Oct. 2006.

[24] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 46:1–46:34, Mar. 2014.

[25] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in C programs," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40, 2007, pp. 356–369.

[26] J. Gummaraju and M. Rosenblum, "Stream programming on general-purpose processors," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38, 2005, pp. 343–354.

[27] Y. Choi, C.-H. Li, D. D. Silva, A. Bivens, and E. Schenfeld, "Adaptive task duplication using on-line bottleneck detection for streaming applications," in *Proceedings of the 9th Conference on Computing Frontiers*, ser. CF '12, 2012, pp. 163–172.

[28] A. Pop and A. Cohen, "A stream-computing extension to openmp," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, ser. HiPEAC '11, 2011, pp. 5–14.

[29] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: Portable stream programming on graphics engines," *SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 381–392, Mar. 2011.

[30] F. Khalid, F. Feinbube, and A. Polze, "Hybrid CPU-GPU Pipeline Framework PDPTA'14."

[31] C. Wagner, "Nullspace approach to determine the elementary modes of chemical reaction systems," *The Journal of Physical Chemistry B*, vol. 108, no. 7, pp. 2425–2431, 2004.

[32] F. Khalid, Z. Nikoloski, P. Tröger, and A. Polze, "Heterogeneous combinatorial candidate generation," in *Euro-Par 2013 Parallel Processing*. Springer, 2013, pp. 751–762.

[33] D. Jevremovic, C. T. Trinh, F. Srienc, and D. Boley, "On algebraic properties of extreme pathways in metabolic networks," *Journal of Computational Biology*, vol. 17, no. 2, pp. 107–119, 2010.