Scalable critical-path analysis and optimization guidance for hybrid MPI-CUDA applications

Schmitt, F.; Dietrich, R.; Juckeland, G.;

Originally published:

August 2016

International Journal of High Performance Computing Applications 31(2017)6, 485-498

DOI: https://doi.org/10.1177/1094342016661865

The International Journal of High Performance Computing Applications
0(0):–

DOI:0
http://hpc.sagepub.com

Scalable critical-path analysis and optimization guidance for hybrid MPI-CUDA applications

Felix Schmitt, Robert Dietrich* and Guido Juckeland
Center for Information Services and
High Performance Computing (ZIH)
Technische Universität Dresden
01069 Dresden, Germany

Abstract

Utilizing accelerators in heterogeneous systems is an established approach for designing peta-scale applications. Today, CUDA offers a rich programming interface for GPU accelerators but requires developers to incorporate several layers of parallelism on both CPU and GPU. From this increasing program complexity emerges the need for sophisticated performance tools. This work contributes by analyzing hybrid MPI-CUDA programs for properties that are based on wait states such as the critical path, a metric proven to effectively identify application bottlenecks. We developed a tool which constructs a dependency graph based on an execution trace and the inherent dependencies of the programming models CUDA and MPI. Thereafter, it detects wait states and attributes blame to responsible activities. Together with the property of being on the critical path we can identify activities that are most viable for optimization. To evaluate the global impact of optimizations to critical activities we predict the program execution using a graph-based performance projection. The developed approach has been demonstrated with suitable examples to be both scalable and correct. Furthermore, we establish a new categorization of CUDA inefficiency patterns ensuing from the dependencies between CUDA activities.

Keywords

GPGPU, CUDA, MPI, wait states, critical-path analysis, performance analysis, performance optimization

* Corresponding author:
Robert Dietrich, Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, 01069 Dresden
Email: robert.dietrich@tu-dresden.de

1. Introduction

The high-performance computing (HPC) community is facing the challenge of peta-scale computing. The pressure to design scalable and energy-aware high-performance systems and applications requires system designers and developers to move from traditional homogeneous to heterogeneous systems, which incorporate accelerators besides standard CPUs. Multiple accelerator options exist, with general purpose graphics processing units (GPGPUs) being currently the most widely adopted technology, especially for energy-efficient computing [1]. For many HPC software developers, the compute unified device architecture (CUDA) is the programming model of choice when designing codes for GPGPU applications. It offers a rich API and is supported by many state-of-the-art software development tools. Nevertheless, CUDA often requires the use of multiple device-side execution streams for peak utilization. Moreover, modern peta-scale systems require the developer to incorporate several parallelization layers, including inter-node communication, to efficiently utilize these machines. Most commonly, MPI is used for this communication due to its proven scalability.

To achieve high performance for computationally intensive and complex applications, critical parts of the code have to be identified and exposed to performance analysis [2]. Today, many tools target either the host (MPI) or the accelerator device (CUDA) for performance analysis, but only few allow a combined analysis of both paradigms. To the best of our knowledge, none allows to analyze the detailed execution dependencies of concurrent CUDA activities, which is required to detect the most-valuable optimization spots in CUDA applications. Therefore, this work focuses on performance analysis for CUDA and its integration with existing analysis techniques for MPI. The contributions of this paper are in detail:

- Categorization of inefficiency patterns for the CUDA programming model
- Detection and quantification of wait states and their direct cause (blamed activity) for hybrid MPI-CUDA programs
- Efficient parallel computation of the critical path based on MPI and CUDA wait states
- Visualization of wait states, blamed activities and critical path in the state-of-the-art visualization tool Vampir
- Graph-based performance projection to determine the impact of CUDA kernel optimization in hypothetical executions

The wait-state-based performance properties and a respective visualization in Vampir guides the user to optimization points and bottlenecks in hybrid CUDA and MPI programs by providing insight into the dependencies between program activities. Our approach uses an application trace in which all appropriate events of a program execution are recorded. It allows the detection of synchronization and communication inefficiencies, which manifest themselves as wait states in the event streams of parallel applications. In the constructed event dependency graph, the critical path can be detected. The critical path is the longest sequence of events through the graph that does not contain any wait states, thereby dominating the total program runtime. It is shown that activities on this path are valuable targets for optimization. The developed tool ranks program activities according to their optimization criticality. Furthermore, we show a method and results for projecting the program execution time using hypothetical runtimes for CUDA kernels. Results are visualized in the state-of-the-art trace viewer Vampir to make the collected information easily accessible to the developer.

This paper is organized as follows: Section 2 presents related work with respect to CUDA and MPI performance tools. Section 3 categorizes inefficiency patterns for CUDA applications. Section 4 presents the pattern detection in hybrid MPI-CUDA application traces using our scalable trace analyzer. Section 5 investigates using the obtained analysis results for projecting the performance of executions with altered function runtimes. The applicability of the introduced methods and tool is presented for two use cases in section 6. Finally, conclusion and ideas for future work are given in section 7.

2. Related work

Software tools like TAU [3] and VampirTrace [4] are able to monitor hybrid MPI-CUDA applications since a few years. The measurement is basically restricted to calls to the accelerator API (e.g. CUDA or OpenCL), kernel execution and data transfer times. Furthermore hardware counters can be gathered for the execution of a kernel. The NVIDIA Visual

Profiler [5] and NVIDIA Nsight [6] provide a more detailed analysis of CUDA programs, e.g. using derived hardware counter information to compute the device's compute and memory utilization, but they are not usable with distributed applications (i.e. with MPI). Moreover, all of these tools are used to perform hot spot analysis, only.

Scalasca is a performance analysis toolset, which focuses on analyzing OpenMP, MPI and hybrid OpenMP/MPI parallel applications [7]. The deployed measurement tool Score-P [8] uses instrumentation to create event traces of program executions. The generated analysis reports can be visualized with CUBE and other third-party profile visualization tools like ParaProf [9]. Scalasca uses parallel replay to detect wait states for MPI programs. It is able to detect direct waiting time that occurs as a result of synchronization or communication inefficiencies as well as indirect wait states that arise from a propagation of those direct inefficiencies through the program execution. After the wait-state analysis, the program is replayed backwards, beginning from the last MPI_Finalize call, to locate the critical path of the MPI execution. This approach scales well since the analysis is performed by as many processes as the analyzed program and the original communication pattern is reenacted. However, it requires to traverse the complete trace information multiple times to generate all analysis data [10]. Among the programming models considered in this work, Scalasca only supports MPI.

HPCToolkit consists of several different tools for performance data recording, analysis and visualization. In contrast to Scalasca, it gathers data using sampling instead of instrumentation. At each sampling point, it collects the current call path along with performance metrics. Combined in a calling context tree, idle regions (i.e. wait states) are detected by identifying blocking routines by name. HPCToolkit then applies a technique called blame-shifting: the blame for causing idleness in one process is attributed to its root-cause or possible suspects [11]. This is possible for MPI applications and can also be used to analyze contention in locking-based models [12]. HPCToolkit is able to apply this blame-shifting approach to CUDA programs by attributing blame to kernels that force the host to wait on their completion or vice versa [13].

While being able to blame certain functions for causing wait states, HPCToolkit does not provide information about functions that are critical for the global execution time. Furthermore, from the information presented by Chabbi et al. in [13], it is not clear if inter-stream dependencies in CUDA can be resolved and there is no official version with CUDA support available to validate it. Finally the accuracy of the measurement is highly dependent on the sampling frequency. Increasing the frequency and thereby the measurement accuracy can result in similar or even higher overheads than synchronous tracing. The advantages and drawbacks of event tracing and sampling have been investigated by Metz et al. in [14].

A reasonable visualization of program traces facilitates the detection of a performance problem which has been shown by Knüpfer et al. in [15]. Nevertheless, it is hard to identify potential optimization candidates that determine the global runtime by investigating process time lines and applying hot spot analysis, only. This work adds the possibility to highlight the critical path and blame in the Vampir time line view, which guides the application developer to optimization spots that are most relevant for global runtime reduction.

3. CUDA dependency patterns

The dependencies between activities running on different parallel execution streams can induce bottlenecks, most often caused by imperfectly balanced communication or synchronization. Many of these problems can be categorized to map to a general inefficiency pattern. Several of those patterns have already been published in literature, e.g. for MPI in [16]. In this section we show a comprehensive categorization of such inefficiencies for CUDA, which includes both host/device and device/device dependencies.

3.1. Definition of parallel event streams

Since both programming models and tools do not share a common terminology, we first define our notion of parallel event streams in the context of trace analysis. An event marks an instantaneous change in the state of an application. This may represent a program entering or leaving a function but can also denominate to the point in time an MPI or CUDA data

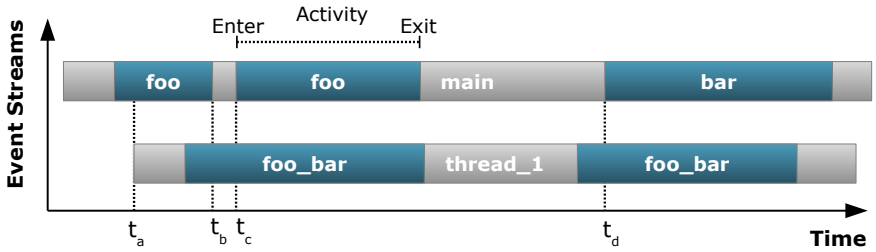

Fig. 1. Parallel event streams: An application trace with two parallel event streams. Each $Enter$ and $Exit$ event has a defined timestamp t. All events of an event stream are ordered by their temporal and semantic relations. Matching events form activities.

transfer is sent or received. This change of state has no duration but is assigned a single timestamp. A trace is the entity of all considered events of an execution along with further information such as hardware-performance counters. An activity is composed from matching enter and exit events (e.g. for a function call). Hence, activities have a start and end timestamp as well as a duration, which is the time difference between both events. Each application execution consists of at least one event stream, an ordered set of events, among which a total ordering relation is defined based on their timestamps and semantic relations. For example, the enter and exit events of an activity must always occur in this causal order. HPC applications are regularly composed of multiple concurrent execution paths, resulting in parallel event streams (see Figure 1). Considering the scope of this paper, parallel events streams can be application threads, the set of CUDA streams on the GPU as well as multiple processes for distributed MPI programs.

3.2. Wait-state analysis

Events in parallel event streams are generally independent. However, dependencies between event streams are imposed by the use of synchronization or communication operations. The undesirable result of load imbalances in the use of such operations is called waiting time or wait state [17]. With the number of event streams per application growing rapidly, it is expected that such load imbalances are among the limiting factors for high scalability [18]. Hence, performance analysis tools must be able to detect those wait states and identify the activities by which they are caused. In MPI, wait states typically occur as a result of the imperfectly timed usage of communication operations, resulting for example in the MPI late sender pattern. In this scenario, one process P_1 calls the blocking MPI_Recv at timestamp t_1 before the sending process P_2 calls MPI_Send at t_2 ($t_1 < t_2$). As a result, P_1 must wait $t_2 - t_1$ before it can start receiving. Waiting time also occurs when collective synchronizations are used, e.g. MPI_Barrier. Here, all processes must wait until the last participant reaches the barrier, thereby often causing wait states in multiple event streams.

3.3. CUDA inefficiency patterns

While patterns causing wait states have been investigated for MPI, no categorization does yet exist for CUDA. In the following, we present our findings for CUDA inefficiency patterns as they are caused by timing imbalances between host and device or between multiple device event streams. Patterns are categorized according to the imperfect usage scenario that is causing the wait state. In section 4 it will be shown how those patterns are detected using our rule-based analysis tool.

Currently, our approach does not take into account performance counters or hardware limitations. This is due to multiple reasons: First of all, this information might not be available in the trace format. In the case of performance counters, several restrictions exist which and how many can be collected in each application. Hence, generating traces including those counters can alter the application timeline and must be considered carefully. Second, restrictions about on-device concurrency and

stricter synchronization behavior can vary between each GPU model and require an extensive database to respect all such constraints. As a result, the applied API model can be considered a subset of all possible dependencies.

Blocking synchronization: The most obvious reason for waiting time is the usage of explicit blocking synchronization API functions. When a host thread calls for example cuStreamSynchronize, the calling event stream must wait until the last device activity that has been enqueued on this event stream before the synchronization operation is completed. The resulting waiting time is the period of time for which device activity and API function overlap. The activity causing the wait state is attributed the blame, a term introduced in [12]. The pattern is called BlameKernelPattern. Another example for explicit blocking operations is cuCtxSynchronize, which waits until all event streams in the requested CUDA context have finished all work. Considering the usage of CUDA events, cuEventSynchronize can be used to wait on the completion of all work preceding its respective CUDA event. Moreover, wait states occur when synchronization between host and device is triggered implicitly by issuing certain synchronous memory operations. These include data transfers (e.g. cuMemcpyHtoD), memory allocations (e.g. cuMemAlloc) and initializations (e.g. cuMemsetD8). Handling of these API functions is analogous to explicit synchronization, hence they are mapped to the same pattern.

Late synchronization: Any blocking synchronizing operation, either explicit or implicit, may be called after all synchronized device activities have been finished. In this case, the wait state is located on the device event stream, which cannot execute any work until the blocking synchronization operation returns. The corresponding pattern is called BlameSyncPattern as the synchronization is blamed for forcing the respective CUDA event stream or even the complete context to idle.

Non-blocking synchronization: The CUDA API allows to asynchronously query the status of a particular CUDA stream or a specific CUDA event by using the functions cuStreamQuery and cuEventQuery. The runtime of these functions is independent from the current state of the queried CUDA primitive and the caller can continue execution regardless of their result. Even though they are non-blocking, their purpose is to notify the host about the device state and enable it to trigger certain execution paths depending on this reported state. Hence, they are regarded synchronization functions, too.

Both cuStreamQuery and cuEventQuery are used to poll the device state. Once the queried work has been completed, CUDA_SUCCESS is returned. Therefore, in a common program execution they might be called several times with a negative result code before success is reported. This last positive query is considered the synchronizing operation for the previously enqueued device activities. In an optimal execution scenario the respective device work would have near-zero execution time. Hence, all query operations occurring while the polled CUDA stream is busy or the CUDA event not yet executed are caused by some executing device activity. As a result, those queries are marked as waiting time if a matching

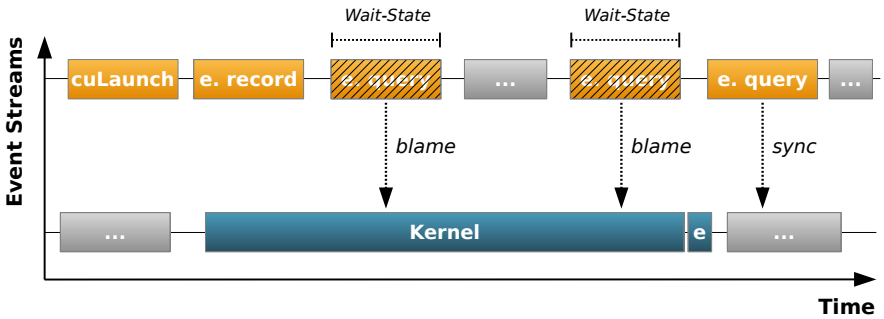

Fig. 2. Non-blocking synchronization: The host polls the state of the device event stream using CUDA event queries ($e.query$). A CUDA event e has been enqueued after launching the device activity ($e.record$). The last poll returns the completion of the tested CUDA event, effectively synchronizing host and device. All other calls to test the device state are waiting time if a successful query is found as they are caused by the runtime of the device activity ($Kernel$).

successful query is found, which effectively synchronizes the device activity. Blame is attributed to the activity keeping the CUDA stream busy or delaying the respective CUDA event. The matching patterns are called EventQueryPattern (see Figure 2) and StreamQueryPattern.

Inter-stream dependencies: Finally, in CUDA programs waiting time can be explained as a result of dependencies between CUDA streams. The null stream is a synchronous, exclusive CUDA stream. Its execution is serialized with that of all other user-created streams. As a result, any other CUDA stream that has outstanding work when the null stream is scheduled can be attributed waiting time. We call this the NullStreamPattern. Furthermore, the execution order of activities in different streams can be controlled using CUDA events together with the cuStreamWaitEvent API function. It forces an event stream to stall work enqueued after this function call until another stream has reached the respective CUDA event (StreamWaitPattern).

4. MPI-CUDA critical-path analysis

Detecting the critical path in application traces has been proven an effective method to identify program bottlenecks and valuable optimization targets [19]. In this section, we show how wait-state patterns can be efficiently quantified in hybrid MPI-CUDA traces. By constructing an event dependency graph, the critical path for an execution can be computed, which is utilized to rate activities by their potential runtime influence. Furthermore, we introduce critical sub-paths to concurrently locate such activities and reduce the time required to compute the critical path.

4.1. Event dependency graph

For computing the critical path in parallel event streams, our tool constructs an event dependency graph (EDG). The EDG is a directed acyclic graph. Each node represents an event and can be identified by a unique, strictly monotonically increasing id number and the id of its event stream. Edges are directed and represent forward-progression in time. Hence, they only connect two events e_1, e_2 that satisfy the condition $t_1 \leq t_2$. Thereby, edges model Lamport's happens-before relation, which describes a partial ordering among all events and can be extended to a global, total ordering as shown in [20]. This allows to traverse events from multiple event streams in a defined order, which is useful for detecting the aforementioned patterns. Moreover, edges are weighted with the duration between two connected event timestamps. For the purpose of critical-path analysis, which partly relies on shortest path algorithms, some edges must be marked as blocking. This is achieved by assigning them infinite weight. For our MPI-CUDA critical-path analysis, we use an hybrid EDG, which is composed from the two sub-graphs for MPI and CUDA, respectively. Each sub-graph is an EDG by itself. Nodes are either MPI or CUDA nodes and are located in one of the two sub-graphs. Similarly, edges are tagged according to which sub-graph they belong. They may be assigned to one graph exclusively or be used in both sub-graphs.

4.2. CUDA pattern detection and graph construction

Our analysis tool is based on OTF trace files [15] that include event records for CUDA and MPI activities of the original program execution. Since we are able to record all required information at runtime using the CUPTI[21] and PMPI[22] interfaces, no source code instrumentation is necessary. This guarantees minimal perturbation of the application run. During measurement, the trace is enriched with OTF key-value pairs that provide the reference information necessary to map CUDA host activities to their respective device event streams and activities. After the application run, the analysis tool concurrently reads the trace and applies its rules to identify inefficiency patterns and compute the critical-path.

The developed tool is an MPI application itself, using the same number of analysis processes as there are MPI processes in the trace. It has been shown by Böhme et al. in [7] that this approach for critical-path detection in MPI applications is highly-scalable even for large numbers of MPI processes. It is important to note that all CUDA dependencies are local to each analyzing MPI process. Therefore, each tool process can handle all child event streams of its assigned application

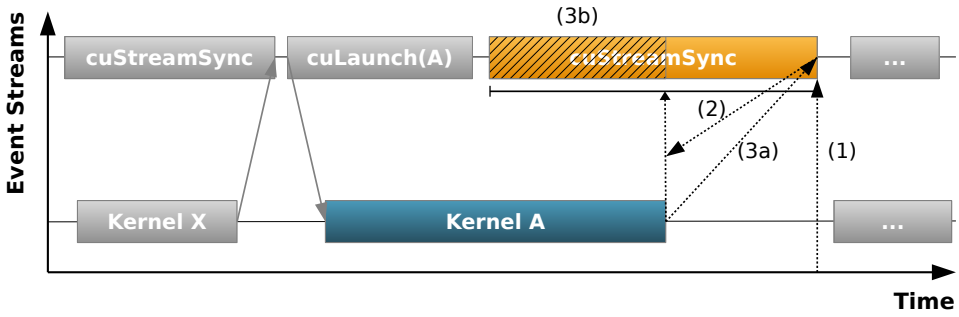

Fig. 3. BlameKernelRule: Detects a synchronization operation that blocks, waiting on a running device activity. Precondition is a synchronization exit node (1) and a not yet resolved activity ($KernelA$) on the event stream referenced by the synchronization (2). The applied transformations are the inserted dependency edge ($3a$) and the blocking wait state ($3b$).

process. These include host threads as well as CUDA streams that have been spawned by the original MPI rank. This has the advantage that the complete CUDA dependency graph for this process is kept local to a single analysis process, thereby enabling local memory access. Each analysis process uses a set of rules to detect the dependencies for CUDA and MPI events and constructs the event dependency graph. Two of these rules for CUDA are explained in more detail in Figure 3 and Figure 4. Rules are applied concurrently by each analysis process to events from all locally processed event streams. The ordering of these local events is determined by their timestamp and causal relation. After all events are processed, the weighted EDG, which includes wait states as blocking edges, has been constructed.

4.3. Identifying optimization-relevant activities

An activity type denotes all instances of a particular function. These may be for example all occurrences of a certain CUDA kernel. Existing profiling tools like gprof or NVIDIA's nvprof [5] perform a hot spot analysis, which focuses on the accumulated exclusive execution time of a certain activity type. This approach is effective to determine code locations where most time is spent but it does not enable to conclude why it is spent there, nor does it allow to detect load imbalances between event streams. Most importantly, it does not differentiate between instances of activities on the critical path and those without potential optimization impact. Hence, we rate activity types according to the accumulated exclusive time they spend on the critical path of the application execution.

Our generated optimization order is based on the two ratings: accumulated exclusive time on the critical path and blame. The first directly represents a function's potential influence on reducing total execution time. The second denotes the indirect,

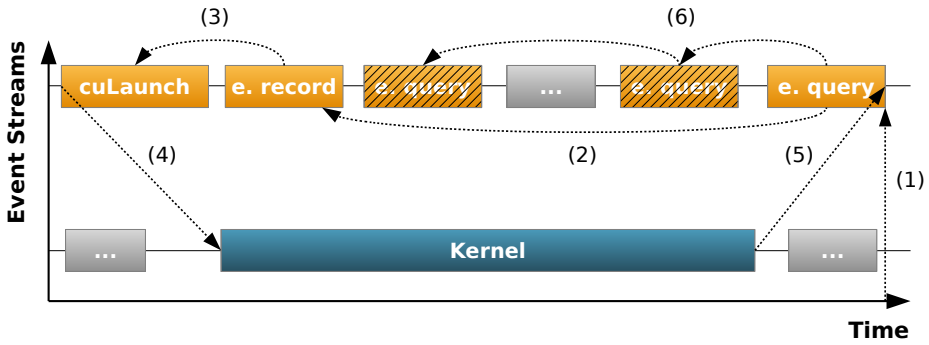

Fig. 4. EventQueryRule: Detects CUDA event queries ($e.query$) polling on a CUDA event of which the last returns its completion. Precondition is the exit node of a successful event query (1). The dependency edge between the device kernel (4) and the last, successful event query is inserted (5). The kernel is identified by the last kernel launch (3) before the matching CUDA event record ($e.record$, 2). Unsuccessful event queries are caused by the kernel runtime and marked as wait states (6).

positive effect any optimization would have towards load balancing concurrent event streams. Blame is attributed according to the amount of waiting time that has been caused by the respective activity and is independent of the critical path. Both ratings are normalized to represent the proportion of time on the critical path and the ratio of an activity type's blame to the overall waiting time in the execution. Ratings are expressed as floating point numbers ranging within $[0, 1]$. Their sum is set as the final rating for a particular activity type. It remains future work to evaluate if ratings should be weighted with different factors to create more precise optimization guidance.

4.4. Using critical sub-paths for efficient analysis

It should be noted that the programming models of MPI and CUDA are independent of each other. Even for CUDA-aware MPI implementations, the MPI library must map GPU-related activities to the CUDA API. In the trace generated for such an application, MPI and CUDA activities are distinguishable. To compute the critical path in the EDG, we can therefore apply a two-phase approach: In the first phase, a parallel reverse replay [10] is used to identify the critical path in the MPI sub-graph. The MPI critical path starts with the call to MPI_Init and finishes at MPI_Finalize. It dominates the execution time for MPI-parallel codes because the program runtime is determined by the longest running MPI process.

A critical sub-path is a continuous part of the critical path in the MPI sub-graph. It is purely located on the event streams that map to a single process, i.e. the CUDA streams created by one application MPI rank. Nodes on the critical path will be called critical nodes. Once all critical nodes of the MPI sub-graph are identified, each analysis process can determine which critical sub-paths are on any of its assigned event streams. In the following, each analysis process only has to compute the critical paths over nodes in the CUDA sub-graph that are on a path between the start and end MPI nodes of a critical sub-path (see Figure 5). The critical path within a critical sub-path is the longest path between its start and end node that does not contain wait states. Hence, critical sub-paths split the CUDA sub-graph even further to multiple, small sub-paths which can be processed concurrently, e.g. by multiple worker threads of one analysis process.

If the application's host part is multi-threaded, critical start and end MPI nodes can be concurrent to CUDA activities on another event stream of this process. In this case it must be decided which CUDA activities belong to a critical sub-path and which do not. Therefore, the following strategy is used to set these splitting points: If an MPI node is the start node of a critical sub-path, all CUDA activities started before this MPI activity must be independent and thus do not belong to the sub-path. If an MPI node is the end node of a critical sub-path, all CUDA activities not synchronized before this MPI activity started must be independent and do not belong to the sub-path. All other CUDA activities are part of the critical sub-path.

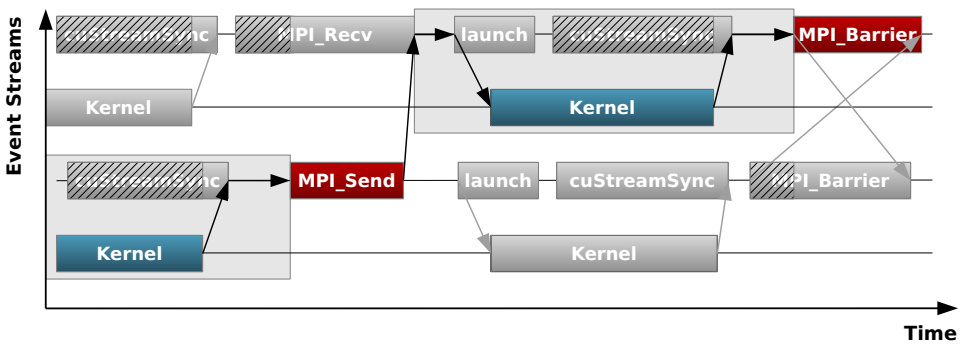

Fig. 5. Critical sub-paths: First, two MPI inefficiency patterns are detected: a Late-Sender pattern (MPI_Recv/MPI_Send) and an imbalanced MPI_Barrier. Then the resulting critical path in the global MPI sub-graph is computed (red intervals). Second, only processes with critical sub-paths (highlighted) need to compute the critical paths in their CUDA sub-graphs (blue intervals). Detected wait states are marked as hatched areas. (Recall that the critical path is the longest path through the execution without wait states.)

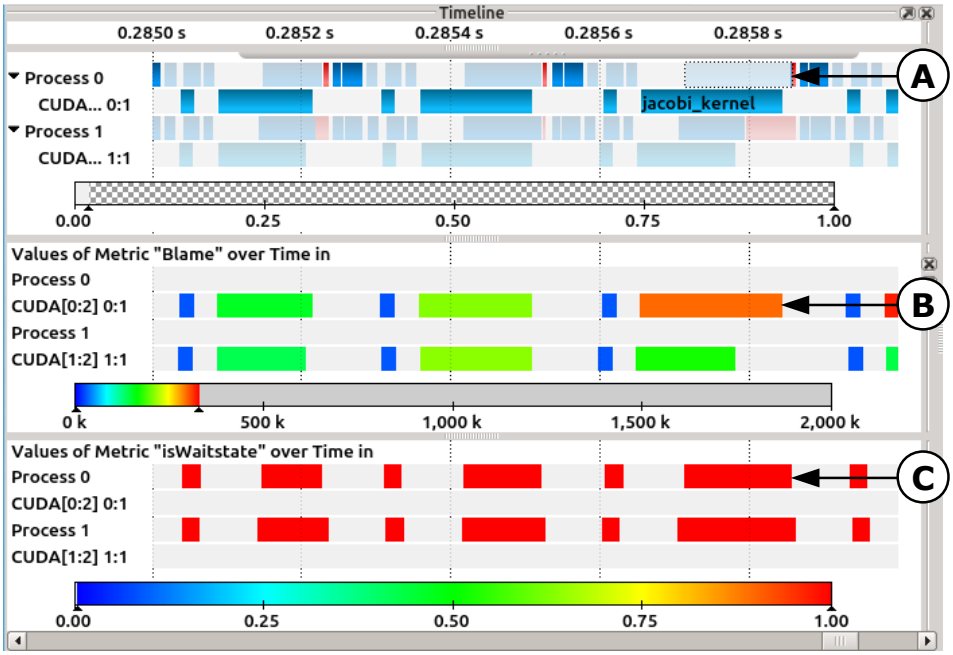

Fig. 6. Vampir visualization: Top-most timeline highlights activities that are on the critical path. Center and bottom counter displays show the scalar blame and binary wait-state information. The selected blocking CUDA memory copy (A) is tagged as a wait state (compare C) and the synchronized kernel is attributed the blame (B).

4.5. Visualization

Visualizing performance analysis results is one of the most helpful but challenging tasks in HPC. We present a basic integration of our generated analysis data into the scalable trace-viewer Vampir. Utilizing the original OTF file along with the derived performance data, a new OTF trace file can be generated as part of the analysis output. This file is loaded into Vampir to visualize analysis results mapped to parallel event streams, represented as timelines in Vampir. To remain scalable, each MPI analysis process creates its own partial OTF files from the assigned event streams. This makes any further synchronization unnecessary. Vampir allows to overlay the timeline for parallel event streams with information from performance counters in the performance radar. To enable this feature the analysis tool tags OTF event records with derived performance counters for each node in the EDG. An example for this visualization is shown in Figure 6. It includes performance counter data representing the critical path, the distribution of wait states and blame attributed to events causing waiting time on other event streams. Since no general CPU activities are processed by the analysis tool, they are omitted in the generated trace file.

5. Performance projection

5.1. Graph-based CUDA performance projection

Based on the EDG (see section 4.1) we developed a performance projection to estimate the effects of optimizations to certain activities. As discussed this graph models all known dependencies between CUDA and MPI events from the event streams in the trace. Assuming that the remaining CPU activities are independent of these events, the knowledge of such temporal relations can be used to predict hypothetical executions. In contrast to the hypothesis verification method used in Scalasca, our graph-based approach does not require to simulate the application's execution in real-time, i.e. CPU activities are not actively replayed by busy-waiting. Instead we recompute the timestamp associated with each graph node. Regarding optimization guidance, this allows to assess the effect of local perturbations on the global program execution.

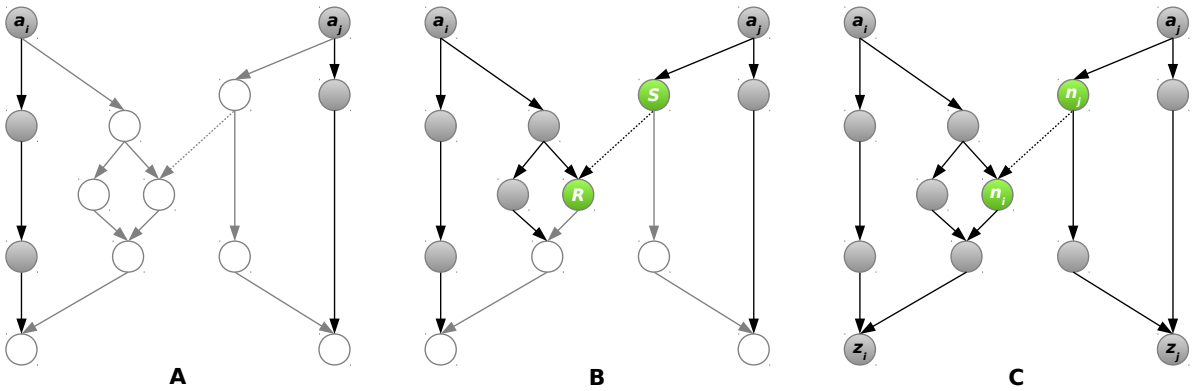

Fig. 7. Graph-based performance projection: A: Each analysis process adjusts as many nodes from a single event stream as possible. B: When there are unresolved input dependencies, processing moves to another event stream. In the case of edges between analysis processes (green nodes), timing information is exchanged via MPI communication operations (dotted line). C: Since all edges represent forward progression in time, the algorithm eventually terminates.

For example, the impact of reducing the runtime of a particular type of CUDA kernel can be estimated both qualitatively and quantitatively. On the one hand, it can be projected if reducing kernel runtime reduces the total program execution time. On the other hand, the ratio between local optimization and global impact can be accurately predicted, given that the graph reflects the actual execution dependencies.

The graph can be traversed in parallel by as many analysis processes as used in the original program execution. Each analysis process handles all event streams that belong to its corresponding application MPI process, similarly to the mapping used during critical-path analysis. A depth-first strategy is used, moving along nodes within one event stream as long as possible. Once a timestamp has been modified, the corresponding node is marked as resolved. While traversing the graph, at each node all ingoing dependencies are tested. If any of these dependency edges is pending, meaning its start node is not yet resolved, the projection continues at another pending node. Should this start node belong to an event stream handled by the same analysis process, a new unprocessed node local to this process is chosen next.

As the graph is distributed and subgraphs are connected via MPI edges, solving of dependencies may require information from another analysis process. In this case, a blocking MPI point-to-point communication is established which waits until the remote start node of the edge has been processed. Likewise, a process sends timing information on an updated MPI node if it has outgoing edges that connect to a remote subgraph. The algorithm is sketched in Figure 7. Since the graph is directed and acyclic, no circular dependencies can exist and the algorithm eventually terminates. The general idea to modify the execution dependency graph is not new. Miller et al [23] proposed to reduce the duration of activities on the critical path to assess the potential impact of their optimization. Mendes and Reed [24] worked on performance prediction and extrapolation using perturbation analysis which altered event timestamps to reflect execution conditions on other systems. However, this is the first time it has been applied to hybrid MPI+CUDA programs.

To further improve the accuracy of performance projection, graph transformations that modify activity timestamps should take into consideration minimal execution times for API functions. When reducing the runtime of a specific host activity such as cuStreamSynchronize due to shortened kernel runtimes, care must be taken not to fall below its minimal execution time. This might require to use a profile that states such execution times for each API function on the target architecture as input to performance projection.

5.2. Performance projection evaluation

Evaluating the performance projection accuracy for CUDA programs using real-world examples is impractical, since most kernels in available applications can not be tuned for a specific speedup. Therefore, a small test tool has been designed.

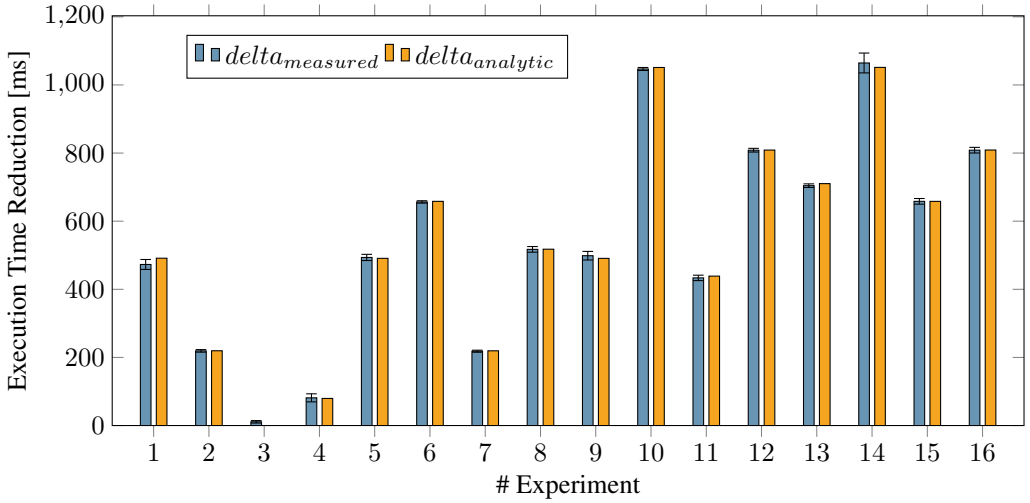

Fig. 8. Projection accuracy for one CUDA stream: Each experiment used a different seed and number of kernels. Blue bars show the average time difference or reduction between runs with and without optimized kernels. Error marks indicate the standard deviation between runs. Orange bars show the time differences projected by the tool. The analysis error is smaller than the deviation between runs. The original total program runtime is between 5.2s and 11.9s.

The tool uses a set of five kernels which execute a certain number of instruction loops each. To simulate optimization, the amount of loops can be controlled during runtime. It has been verified that reducing the number of loops by 50% accurately reduces the kernel runtime for the tested kernels. When starting the tool, it reads the number of CUDA streams, the number of kernel to execute, the initial seed for the pseudo-random number generator and the optimized kernel from the command line. Each round, a kernel is chosen randomly and launched on a stream. In some rounds, a randomly determined stream is synchronized. For each individual experiment, the test tool is run with a specific seed once with and once without kernel k_1 optimized by 50% ($time_{opt}$ and $time_{orig}$). Each experiment is repeated ten times on an NVIDIA Kepler K20X GPU. The original total application runtime for an experiment has been between 5.2s (#1) and 11.9s (#16).

Both experiment runs have been instrumented using VampirTrace to record the respective application execution trace. Time differences between each two executions is computed as $delta_{measured} = |time_{orig} - time_{opt}|$. For each experiment run, the trace file without optimized kernels is used as input for the performance projection which predicts the optimized runtime. The time difference between predicted execution time ($time_{analytic}$) and original time is denoted as $delta_{analytic} = |time_{orig} - time_{analytic}|$. Figure 8 shows $delta_{measured}$ and $delta_{analytic}$ for 16 experiments with one CUDA stream. The standard deviation over all original runtimes of one experiment is $\sigma_{measured}$, indicated by the error bars.

It can be seen in most experiments that both measured and predicted execution time reduction are notably larger than the deviation between runs. Exception is experiment three for which $delta_{analytic}$ is zero. In this experiment, the optimized kernel k_1 is not executed at all due to the used random seed. Hence, the analysis tool predicted no runtime reduction and $delta_{measured}$ is a result of runtime variations between executions of the test application. Moreover, the performance projection error $|delta_{measured} - delta_{analytic}|$ is smaller than the standard deviation for all other experiments. This concludes that the optimized runtime can be predicted with high accuracy for single CUDA stream applications. For experiments with multiple available CUDA streams, the stream is chosen randomly, too. Moreover, after each kernel execution it is determined which stream is to be synchronized or if no synchronization operation is called at all in this round. Figure 9 shows results for experiments with multiple CUDA streams per run. It can be seen that projection accuracy decreases when more CUDA streams are used concurrently.

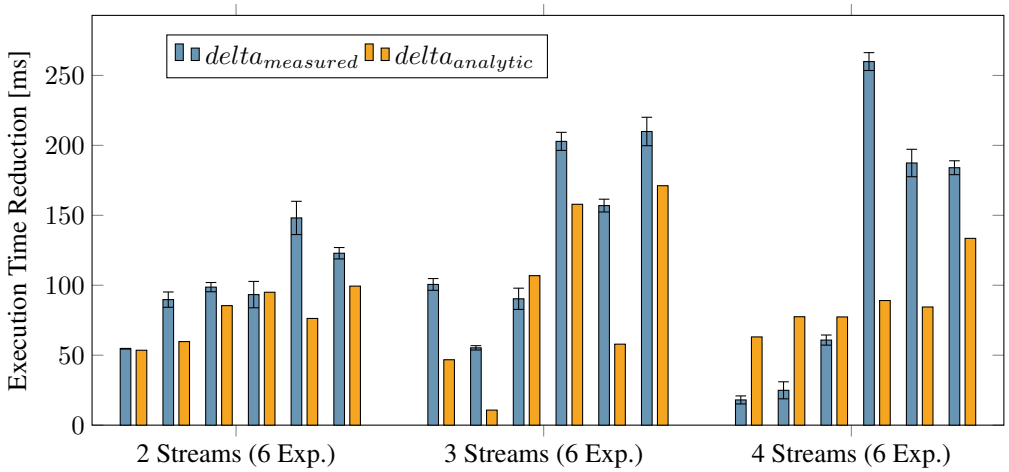

Fig. 9. Projection accuracy for multiple CUDA streams: The difference between measured and predicted ($delta_{analytic}$) execution time reduction increases the more concurrent streams are used. With multiple CUDA streams and independent kernels, the GPU scheduler is free to reschedule kernel executions as the device utilization changes during the optimized run. To accurately predict this behavior, the analysis would have to capture the dynamic scheduling and resource utilization effects of concurrent kernels. The original total program runtime is between 0.5s (2 streams) and 1.9s (4 streams).

The respective average errors of $delta_{analytic}$ over $delta_{measured}$ are 19% (two streams), 42% (three streams) and 51% (four streams), normalized to the maximum of both durations. This is a result of the fact that the model does not account for any resource requirements and constraints other than time. CUDA capable devices feature a certain limited number of CUDA cores, grouped in SM(X) multiprocessor units. CUDA threads are grouped in warps and blocks, where each of the latter is assigned to a specific multiprocessor. NVIDIA GPUs of the latest Kepler generation are capable of concurrently executing many blocks from the same kernel and multiple kernels from different CUDA streams as long as free multiprocessors are available. Hence, independent kernels can run fully parallel only if enough resources are available for both. Similarly, optimizing one kernel can reduce the runtime of concurrent kernels as multiprocessors become available earlier. The performance projection model on the other hand assumes that all kernels which do not have execution dependencies such as a pending CUDA event are ready to be executed fully concurrently without affecting each other. This shortcoming of the dependency model results in miss-predicted start and execution times for kernels launched to different CUDA streams. The effect increases when more independent kernels are launched by the application to multiple, independent CUDA streams in parallel. Hence, a more sophisticated dependency model that takes resources into consideration is required.

6. Use cases

To demonstrate the semantic correctness of our hybrid critical-path analysis, an existing implementation of the Jacobi method is used. The source code has been provided by NVIDIA Application Lab at Jülich [25]. In this example, the Jacobi method iteratively solves the discrete Poisson equation with Dirichlet boundary conditions. The use case is implemented using CUDA and utilizes MPI for parallelization across multiple nodes. It has been chosen since it allows to statically load balance work between host and device by defining the offloading ratio for the GPU.

A low offloading ratio results in an under-utilized GPU, which runs idle before the CPU computation reaches the synchronization points. Hence, mostly CPU activities are expected to be on the critical path. When the offloading ratio is high, i.e. most of the work is computed using CUDA, we expect that the GPU kernels are on the critical path and blame is mostly assigned to CUDA kernels, because the host is running out of work and synchronizes with the GPU before the respective kernels finish execution.

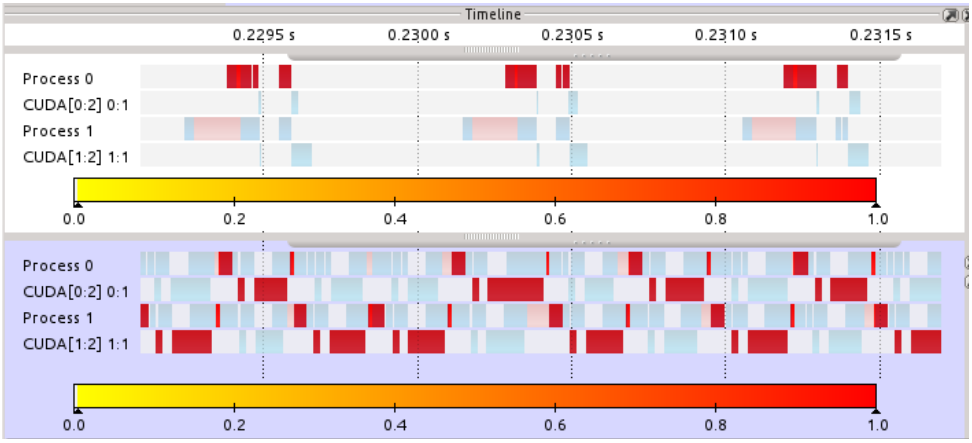

Fig. 10. Vampir compare view of Jacobi example: Only a snippet of each result trace is shown. In the example with 10% work offloaded to the GPU (top timeline), the critical path is entirely on CPU event streams (highlighted red). For the example with 90% offload ratio (bottom timeline), the critical path changes between CPU and GPU activities.

Figure 10 shows a Vampir compare view of the two traces generated by the analysis tool. The critical path over CUDA and MPI activities is marked in red in both timelines, highlighting the expected analysis result. General CPU activities are omitted in the trace. In the case with little work offloaded to the CUDA capable GPU, most work is done by the CPU. When the device streams are synchronized from their respective host streams, kernel executions have already been finished and the host does not need to wait on the device. As a result, the critical path is solely on host event streams as it can be seen in the top trace. In comparison, the bottom trace shows analysis results for 90% work offloaded to the GPU. Here, the total application runtime is determined by the kernel executions on which the host streams must wait during CUDA synchronization operations. This leads to the displayed effect that the critical path of the application moves between host and device activities of one process, in addition to moving across MPI processes when such communication occurs.

Activity ratings are computed from the fraction of time an activity is on the critical path and the total fraction of waiting time it caused. For the case with little work offloaded to the device, all kernel activity types are assigned zero ratings since they do not contribute to the critical path nor do they cause any wait states. Table 1 shows the respective activity ratings generated during the analysis for the high-offload example. Here, the compute-intensive jacobi_kernel is rated with the highest optimization priority. Concluding, it can be seen that the results have been computed as expected by the developed analysis tool.

We evaluated the scalability of our analysis tool using the CUDA accelerated version of high performance Linpack, HPL CUDA [26]. It solves a random system of dense linear equations and is the primary HPC evaluation software for

Table 1: Ratings of activity types in the Jacobi example for 90% work offloaded to the GPU.

Activity (all instances)	Critical Path [%]	Blame [%]	Rating
jacobi_kernel	40.69	35.34	0.7603
cuMemcpyDtoH_v2	30.10	5.6	0.3570
MPI_Barrier	0.0	35.62	0.3562
copy_kernel	5.04	9.59	0.1463
MPI_Allreduce	0.0	12.78	0.1278
cuMemcpyHtoD_v2	10.15	0.0	0.1015
cuLaunchKernel	3.63	0.0	0.0363

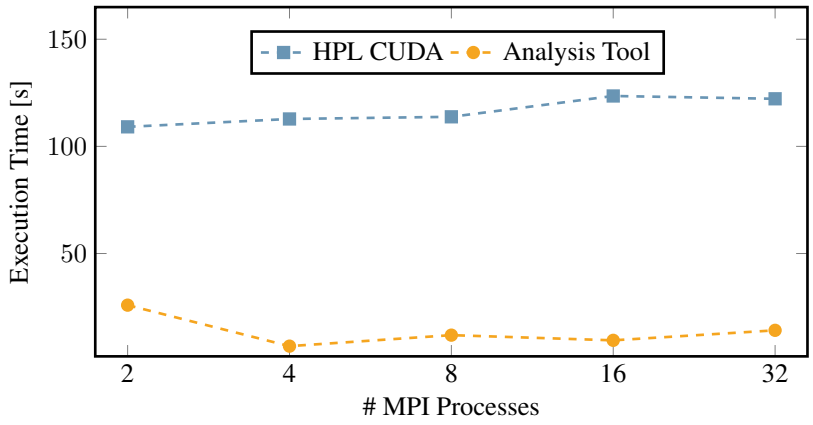

Fig. 11. Comparison of HPL CUDA and analysis tool runtimes for different numbers of MPI processes. Matrix size for HPL CUDA has been chosen so that its execution time stays approximately constant. The analysis tool scales the same as the original application.

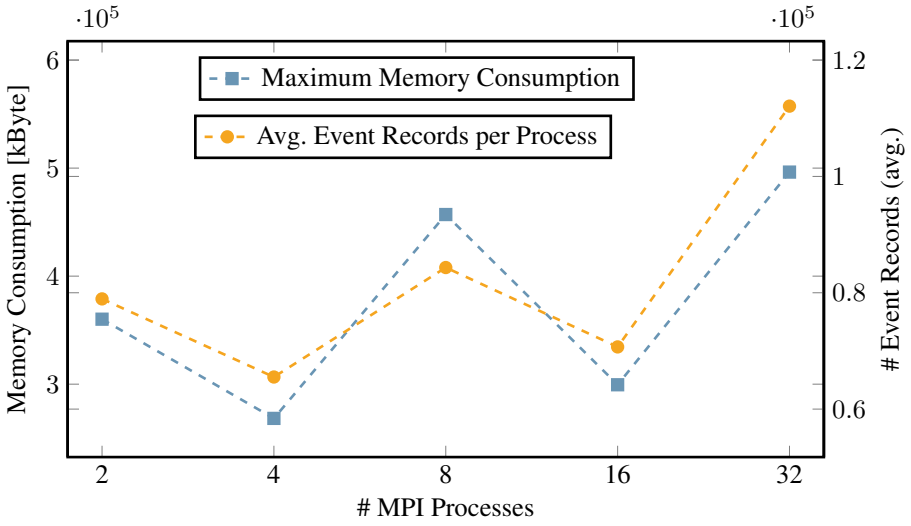

Fig. 12. Relation between memory consumption and average number of processed events for HPL CUDA: The figure shows the maximum memory consumption per analysis process in relation to the average number of event records processed by any process. The two strongly correlate for this example, meaning that the memory consumption is determined by the number of processed event records.

testing the peak performance of large systems, mainly because it is known to exhibit very good scalability. MPI is used to distribute the computation across multiple nodes and both CPU and GPU are utilized for computation on each node. OpenMP host-side parallelization has been disabled for this test as the current state of the tool does not properly support this programming model. We mapped each process to a dedicated node with a single NVIDIA K20X GPU. The input to HPL CUDA has been chosen so that the runtime stays approximately the same when increasing the number of MPI processes. Figure 11 shows the resulting execution times for both HPL CUDA and the analysis tool for up to 32 processes. It can be seen that the tool runtime scales similar to the execution time of the analyzed application when modifying the number of MPI processes. Since the applied parallel reverse replay technique for identifying the critical path in the MPI sub-graph is known to scale to very large numbers of processes, we expect our combined MPI-CUDA analysis to scale equally well.

The total memory consumption per process depends on the number of processed events in the trace, i.e. CUDA and MPI records. For each event record an internal Node object is allocated along with other data, e.g. dependencies and statistical information. Moreover, it is influenced by the complexity of the detected patterns since complex patterns require to store a larger number of dependencies between the events. Each dependency is stored as an internal Edge object and references are inserted into the respective lists for in- and outgoing edges of each Node. Figure 12 shows both the number of event

records in the trace and the maximum memory consumption among all analysis processes for different numbers of MPI processes. It can be clearly seen that both functions are strongly correlated. Additionally, the number of event records that need to be analyzed during analysis does affect the total execution time of the analysis, too. The tool runtime is primarily determined by the analyzed application. However, comparing the execution time of the analysis tool in Figure 11 to the memory consumption data highlights that the latter also influences the tool runtime.

7. Conclusion and outlook

We presented an analysis approach that computes the critical path for in-depth analysis of hybrid MPI-CUDA applications. Using non-intrusive tracing in connection with a dependency model for CUDA and MPI operations, inefficiency patterns and wait states are detected and quantified. The developed tool recognizes these patterns using a set of rules, which makes the approach easily extensible. Our rules allow to identify waiting time and their direct cause. In the constructed distributed event dependency graph, the critical path is detected to determine activities that are valuable targets for optimization. In contrast to traditional hot-spot analysis, we rate activities according to their potential for improving the total program runtime. Besides generating an ordered list of optimization targets, analysis results are presented in the state-of-the-art trace viewer Vampir, which makes the information visually accessible to the user.

All applied methods are parallelized using MPI to guarantee scalability, for instance by integrating parallel reverse replay to identify the MPI critical path. Besides MPI parallelization we introduced critical sub-paths, which allow to efficiently and concurrently compute the critical path in the CUDA sub-graph of hybrid traces. The applicability and scalability of the presented approach has been demonstrated with two suitable examples. Whereas our analysis deals with CUDA's asynchronous execution model it remains a challenge to effectively process non-blocking MPI communication.

We also investigated the use of the dependency graph to project program runtimes for executions with modified CUDA kernel durations. While results are very precise for applications utilizing only a single CUDA stream per GPU, accuracy quickly decreases as multiple streams are used concurrently. This stems from the fact that the prediction model only considers runtimes but fails to take other resource constraints into account.

The rating of optimization targets could benefit from including hardware performance counters in the analysis. GPU kernel counters include information such as achieved device occupancy, branch efficiency or the number of cache misses. These can be used to create an estimate of the efficiency of a particular CUDA kernel. For example, kernels that significantly under-utilize the device or show a high cache miss ratio are more likely candidates for optimization than those that display perfectly tuned memory access patterns. These data can be considered when creating the rating for kernels to point the user attention to even more likely optimization candidates.

Another interesting research aspect raises the question if it can be supported by the tool to specify new rules as formal descriptions for existing and additional programming models. Such specifications must be able to represent both syntactic and semantic properties of the programming model. The former being for example the list of valid API function names while the latter are the requirements and dependencies imposed by a certain activity occurring in the trace. During tool runtime, those descriptions could be parsed, converted to executable rules using a generic rule generator and applied to the analyzed parallel event streams. This could be achieved efficiently by identifying similarities among related programming models that allow to map their syntax and semantics to more abstract dependency patterns.

References

[1] The Green500 List, November 2014;. Available from: http://www.green500.org/lists/green201411.

[2] Manferdelli JL, Govindaraju NK, Crall C. Challenges and Opportunities in Many-Core Computing. In: Proceedings of the IEEE. vol. 96 Issue 5. Microsoft Corp., Redmond. IEEE; 2008. p. 808–815.

[3] Mayanglambam S, Malony AD, Sottile MJ. In: Performance Measurement of Applications with GPU Acceleration using CUDA. vol. 19 of Advances in Parallel Computing. IOS Press; 2010. p. 341–348.

[4] Dietrich R, Ilsche T, Juckeland G. Non-Intrusive Performance Analysis of Parallel Hardware Accelerated Applications on Hybrid Architectures. In: 39th International Conference on Parallel Processing Workshops. ICPPW. IEEE; 2010. p. 135–143.

[5] Profiler User's Guide; 2014. Available from: http://docs.nvidia.com/cuda/profiler-users-guide/, last visit Sep. 22th, 2015.

[6] Nsight Visual Studio Edition 3.2 User Guide; 2013. Available from: http://docs.nvidia.com/nsight-visual-studio-edition/3.2/Nsight_Visual_Studio_Edition_User_Guide.htm, last visit Sep. 22th, 2015.

[7] Geimer M, Wolf F, Wylie BJN, Erika Abraham DB, Mohr B. The Scalasca Performance Toolset Architecture. Concurrency and Computation: Practice and Experience. 2010;22(6):702–719.

[8] Mey Da, Biersdorf S, Bischof C, Diethelm K, Eschweiler D, Gerndt M, et al. Score-P: A Unified Performance Measurement System for Petascale Applications. In: Bischof C, Hegering HG, Nagel WE, Wittum G, editors. Competence in High Performance Computing 2010. Springer Berlin Heidelberg; 2012. p. 85–97.

[9] Performance Research Lab. ParaProf User's Manual; 2010. Available from: http://www.cs.uoregon.edu/research/tau/docs/paraprof/, last visit Sep. 22th, 2015.

[10] Böhme D, Wolf F, Geimer M. Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications. In: 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW). IEEE; 2012. p. 2538–2541.

[11] Tallent NR, Adhianto L, Mellor-Crummey JM. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE; 2010. p. 1–11.

[12] Tallent NR, Mellor-Crummey JM, Porterfield A. Analyzing Lock Contention in Multithreaded Applications. SIGPLAN Notices. 2010;45(5):269–280.

[13] Chabbi M, Murthy K, Fagan M, Mellor-Crummey J. HPCToolkit: A Tool for Performance Analysis on Heterogeneous Supercomputers. San Jose, CA; 2013. Talk at Nvidia GPU Technology Conference (GTC).

[14] Metz E, Lencevicius R, Gonzalez TF. Performance Data Collection Using a Hybrid Approach. SIGSOFT Software Engineering Notes. 2005;30(5):126–135.

[15] Knüpfer A, Brunst H, Doleschal J, Jurenz M, Lieber M, Mickler H, et al. The Vampir Performance Analysis Tool-Set. In: Resch M, Keller R, Himmler V, Krammer B, Schulz A, editors. Tools for High Performance Computing. Springer Berlin Heidelberg; 2008. p. 139–155.

[16] Wolf F, Mohr B. Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture. 2003;49:421–439.

[17] Meira W Jr, LeBlanc TJ, Almeida VAF. Using Cause-effect Analysis to Understand the Performance of Distributed Programs. In: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools. SPDT '98. New York, NY, USA: ACM; 1998. p. 101–111.

[18] Daly J, Hollingsworth J, Hovland P, Janssen C. Tools for Exascale Computing: Challenges and Strategies. Office of Science, U.S. Department of Energy; 2011.

[19] Yang CQ, Miller BP. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In: 8th International Conference on Distributed Computing Systems. IEEE; 1988. p. 366–373.

[20] Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM. 1978;21(7):558–565.

[21] CUDA Toolkit Documentation — CUPTI; 2014. Available from: http://docs.nvidia.com/cuda/cupti/index.html, last visit Sep. 22th, 2015.

[22] MPI: A Message-Passing Interface Standard, Version 2.2; 2009.

[23] Miller BP, Clark M, Hollingsworth J, Kierstead S, Lim SS, Torzewski T. IPS-2: The Second Generation of a Parallel Program Measurement System. Transactions on Parallel and Distributed Systems. 1990;1:206–217.

[24] Mendes, Celso L . Performance Prediction by Trace Transformation. In: Fifth Brazilian Symposium on Computer Architecture; 1993. .

[25] Adinetz A, et al . NVIDIA Application Lab at JSC. In: inSiDE. vol. 11; 2013. .

[26] Fatica M. Accelerating Linpack with CUDA on Heterogenous Clusters. In: Proceedings of 2nd Workshop on General Purpose Processing

on Graphics Processing Units. GPGPU-2. ACM; 2009. p. 46–51.