Smart Traces: Trace Visualization in Parallel Dataflow Frameworks

Category: Research

Abstract—After hitting the GigaHertz barrier in the past decade the standard approach for increasing program performance has been parallelism, in an ever increasing diversity of hardware and software platforms. Each of those platforms introduces unknown variables that affect the program’s performance, and even correctness, in ways different (or less common) than with sequential programs. Being able to understand how the program execution behaves is fundamental to make optimal use of the existing platforms. Existing tools can summarize results, but hardly can manage a large amount of execution data with complex relations. In this work we investigate the requirements to make a visualization of trace data extracted from executions of programs modeled as dataflows to enable easier analysis from the visual representation of the data. We propose Smart Traces, a concept that aims at creating visualizations for trace data that serve as a supporting tool to analyse performance. We demonstrate with examples the power of visualizations that we can create to address specific questions formulated about the analysis of the data, with emphasis to parallel dataflow traces.

Index Terms—Software visualization, graph/network data, time series data, coordinated and multiple views.

1 INTRODUCTION

Parallel computation in its many different forms is essential to improve performance of computational tasks. Such computation power was the target of extensive research in the late eighties and early nineties, where seminal work in the theory of parallel computation was developed. However, its widespread use only become common recently, with the rise of multi-core CPUs, many-core GPUs, and computer clusters at affordable prices. Each of these configurations have several aspects that impact performance, and leveraging the benefits of parallel computation to the extreme is still a challenging task.

An important component of this problem is the post-analysis of computational traces generated upon execution of a parallel program. A parallel trace records a time-series of information about resource utilization and activation procedures during program execution. Trace analysis and visualization was discussed in early work in the past [11, 17, 9, 31], which identified the value of visualization techniques to help understand the complex interplay of information in trace data.

The most important visualization tool often employed consists of a Gantt chart, which corresponds to a 2D time-series graph that displays in which processing unit a task is executing in a given instance of time. The development of information visualization techniques along the years combined with the increase and diversity of parallel computation power available today allows the visualization and analysis of parallel traces to be revisited under a new perspective.

In this work we introduce Smart Traces, a new concept that aims at producing insightful visualizations of parallel trace data generated in dataflow systems. The widespread availability of dataflow systems and its growing application in scientific applications using parallel computation makes them an important target for performance improvement. In particular, our main motivation for this work was to create a tool that helped us better understand and debug a parallel dataflow framework for heterogeneous systems we are currently working on. The trace visualization tool [16] we used failed to capture some aspects of the trace data, in particular its relationship with dataflow systems.

Smart Traces was designed to address several challenges raised by this analysis. Trace data encodes complex relationships among computational resources in time (e.g. processors, tasks, memory transfers, etc), which need to be grouped and visualized in different ways, sometimes in isolated visualizations, but most frequently as coupled visu-
We designed a trace representation format augmented with dataflow semantics to allow trace visualization and analysis. Furthermore, while it is easy to visually accommodate the individual display of trace data for dozens of processing units, the growing number of parallel execution units available increases the amount of information to be displayed. Therefore, strategies that collapse time-series data into meaningful representations are required (e.g. Theme Rivers[10], Stacked Graph[4], Edge Bundles[12] or Word Lines[33]).

We customized the supporting information visualization toolbox offered by the DEFOG system [19] to process our trace format, and allows the trace visualization to be coupled with the dataflow processing being executed. A flexible mapping of infovis tools to trace data is used, which allows for the creation of an arbitrary number of visualizations. Such mappings, called Smart Traces, represent one view of the trace data. As a result, we are able to produce and interactively modify diverse trace visualizations. In summary, the main contributions introduced in this work are outlined below:

- A trace data format augmented with dataflow information that is suitable for mapping information visualization techniques;
- The concept of Smart traces, which correspond to views of dataflow trace data constructed upon a programmable information visualization toolbox;
- A collection of predefined smart traces suited for the analysis of dataflow trace data, which allows for immediate trace visualizations to be constructed directly from the trace data.

2 RELATED WORK

There is a diverse literature that discusses several aspects related to parallel trace visualization. Although several of this work address common topics, we summarize them in separated sections to better explain the main contribution of each work with respect to this problem.

Visualization of Parallel Systems

The Paragraph system [11] was one of the earliest work to identify the importance of visualization techniques to understand trace information in parallel systems. It raises the importance of creating an animation that reflects the dynamic behavior of trace, but more importantly, sets the foundations for desired features in such a system (e.g. utilization, communication, task, and application-specific displays). To our knowledge, it is the first work that uses Gantt charts (created for industrial labor management purposes[7]) to visualize time-series data of resource against task utilization. Much of the ideas proposed in our work relate to the foundations established by this seminal work. Several work [17, 9] that followed expanded the ideas of Paragraph and emphasized user interface aspects that allow for multiple views of the data. Thomas and Ueberhuber [31] defined design goals, techniques and software introduced up to that moment (middle of the nineties).

The introduction of message-passing protocols led to specifically-designed visualizations to understand the performance of such protocols. The Viper[26] system uses a standard set of views to inspect a given aspect of the parallel program state: animation, space-time or variables view. It uses a different parallel programming paradigm, called Mona Lisa, that allows data transfer to occur only when needed, with trace messages generated (start or termination of a primitive call, sending or reception of a message). A system with focus more into data-parallel visualization than performance is described in [32], with emphasis on a system that could handle traces interactively. MPI trace visualization is first discussed in [15] and further elaborated some years later in the work of Wu and colleagues at IBM [35, 34]. Their work encompasses a unified trace environment that allows both user and system activities to be monitored. Visualization relies on the use of different Gantt charts to evaluate thread and CPU activities.

With the popularization of parallel computing due to multi-core CPUs, many-core GPUs and affordable clusters, several work on the subject surfaced recently, with special emphasis on the efficient storage of trace data. Tracevis [24] was designed as a tool to better understand micro-architectural simulation data. Novel to this approach is the search for trace regions that match some criteria, as well the ability to annotate traces with persistent information, which allows for collaborative trace analysis. The Vampir NG system [16] offers a client-server solution coupled with a compressed data structure to handle large trace data, as well as it uses parallel processing to improve the trace analysis performance. A similar idea of augmenting trace information with meta-data is described in [20] with the purpose of obtaining analytical performance models of MPI programs. In [22] is described Data Flow Tomography, a system that uses virtual machines with special instructions that tag bytes of memory, and thus keep track of all data movement and usage. application-specific semantics. The DIMVisual model [29] allows the integration of trace data in a distributed systems, with subsequent treemap [28] and 3D visualization [27] approaches for grid monitoring and its respective network topology. A recent work [8] describes trace visualization techniques for flow analysis in applications running on IBM’s Cell processor and NVIDIA GPUs using CUDA. Their trace analysis mostly uses Gantt charts augmented with links between tasks to show interconnections.

Visualization Techniques for Time-series Data

Common visualization of trace data uses summary graphs, such as histograms of function call counts, pie chart distribution of time spent in each module, etc. This information continues to be important, but they do not capture important aspects or anomalies in resource utilization that occur during the execution. For this purpose, Gantt charts[7] offer a display of a time-series of information with individual processor utilization. A standard Gantt chart becomes too dense and less useful when the number of resources is too large, or when events are too small and/or too numerous to fit the display area.

There are several techniques in the literature that address more compact or more expressive visualization of time series data, and a complete survey is beyond the scope of this work. Approaches like Theme Rivers[10] or Stacked Graphs[4] compact information using semantic information of the time-series to adjust the thickness of the stack with respect to a given baseline. The Hierarchical Edge Bundles [13] applies the idea of edge bundles [12] to compact trace information, in particular it aims at capturing important activation call relations. A study on the graphical perception that compares different time-series visualization is described in [14], which identifies pros and cons of each method with respect to certain tasks. In addition to the standard Gantt chart representation, in this work we use Stacked Graphs to display Gantt chart information, which becomes specially important to summarize information and quickly identify anomalies for traces composed of a large number of processors (we report results using machines with up to 256 cores).

Coordinated and Multiple Views

Filtering, grouping and summarizing allows for compacting information into a smaller display area while still showing relevant information. However, to capture the different aspects of data, in particular the complex interplay of trace information, it becomes necessary to combine multiple visualizations in a common linked (or coupled) view. There are good introductory texts [2, 25, 1] on the subject of creating Coordinated and Multiple Views (CMV). The fundamental ideas of CMV encompass an adequate representation of data, tasks and analysis tools that allow for creating linked visualizations. Examples of systems developed using the CMV strategy include [3, 21] among others. Recent work in the area include the Matchmaker [18] visualization technique that groups multi-dimensional datasets, the Word Lines [33] visualization technique to control multiple simulation runs, and behaviorism [6] which defines strategies to create visualizations for dynamic data and its interconnections.

We strongly believe that the CMV philosophy is essential for the visualization and analysis of trace data. Our proposal relies on the information visualization toolkit and CMV infrastructure offered by the Defog [19] system to construct trace visualizations. The two major components of DEFOG are a drawing environment, where the user can interact with graphical elements similarly to popular drawing packages, and an interactive programming environment, where complex
operations can be executed by using Python code. A collection of Python programs is readily available, that manipulate the graphical elements to construct visualization, while still referencing the original data objects. This gives flexibility to interpret and manipulate the underlying data. Examples as given in the supplemental material.

3 PARALLEL DATAFLOW EXECUTION AND INSTRUMENTATION

In this section we define the characteristics of a parallel dataflow system, and how the execution trace data is collected.

3.1 Parallel Dataflow Model

A dataflow is a directed graph where each node corresponds to a module whose execution depends only on its inputs, and interaction between modules happens by data exchange. Each input parameter and output result in a module is denominated a “port”. Output ports are connected to input ports, defining the topology of the pipeline of the dataflow as edges of the graph. An input triggers a series of computations and data communication down the pipeline.

In a sequential execution environment the dataflow is topologically sorted, which forces a module to only be executed after its dependencies have executed. In parallel systems it is desirable to have multiple threads or processes to execute modules in parallel. There are multiple approaches for implementing a parallel dataflow architecture as mentioned before, but they require non-trivial amount of coordination among threads and processes to make optimal use of resources. This coordination is usually performed by a scheduling entity, responsible for assigning tasks to modules, and for starting their execution. Depending on the program requirements, the scheduler can offer some guarantees, such as order of execution, real-time constraints, maximum number of resources to be allocated, etc, at the cost of a more complex and time-consuming scheduler implementation.

3.2 Code Instrumentation

The execution trace consists of time-stamp events collected by code instrumentation at specific points of interest. Code instrumentation can be performed at different levels-of-detail, and in the particular trace analysis we want to perform our attention lies to events related to dataflow execution. Automated tools for code instrumentation (e.g. DTrace[5], Vampir[23], TAU[30]) offer a more fine-grained collection of events to construct visualization, while still referencing the original data objects. This gives flexibility to interpret and manipulate the underlying data. Examples as given in the supplemental material.

Fig. 2. Smart Trace Architecture. A parallel dataflow executes multiple tasks, and the trace execution is collected into a Smart Trace collection, which consists of trace data augmented with dataflow information. The smart trace composition and exploration relies on standard information visualization tools or predefined constructions to define a view (Smart Trace) of the trace collection.

In addition to the events mentioned above we store additional metadata that represents an entire part of the execution of the dataflow network and the data movement. The execution and data transfer in dataflow systems is managed from specific entities, making it easier to map events to the task that triggered them, as well as to register events since the sender/receiver information is stored in the system explicitly. Edges in the dataflow graph define channels where messages are exchanged, with senders and receivers corresponding to executions running at the modules. Trace events have a dataflow task associated, thus allowing for later grouping of events based on input.

4 SMART TRACES

In this section we introduce the Smart Trace abstraction and describe how the user creates customized visualizations of trace data. The concept is implemented in DEFOG, making full use of its dynamic environment, where code and data can be created and updated at any time to create linked visualizations.

The process of creating a Smart Trace visualization proceeds as illustrated in Figure 2. First, the execution trace of a parallel dataflow is recorded using code instrumentation. Trace data is augmented with dataflow information, resulting in what we call a Smart Trace collection. This data is imported into a customized version of the Defog system, in which the user can analyse the data by creating views of the data, called Smart Traces. The user create visualizations using the DEFOG toolbox of information visualization techniques, but can also directly import the data into Smart Traces generated previously.

4.1 Smart Trace Collection

A smart trace collection is a collection of trace data augmented with dataflow information, which is stored as a list of Python objects with explicit relations defined using object references. Object abstractions
were specifically designed for the dataflow trace analysis we want to perform, but other abstractions can be used to create different smart trace collections in other applications. The object abstractions are:

- State change: actual raw data explicitly registered in the execution trace. In addition to start and finish times for each module execution, it references the computation unit where it was executed (stream), predecessor and successor events according to both the pipeline and the stream, task being processed, and application-specific information (e.g. local memory used);
- Stream: associated to a system thread, process, or CPU (when thread affinity is set). Contains start and finish times, references to events executed underneath it, and the trace session;
- Trace Session: global execution of the program, composed of per-application metadata;
- Task: execution of one input across the whole pipeline. It contains attributes such as start and finish times, as well as any application-specific attributes describing the initial input and final output of the pipeline;
- Module: one pipeline stage, with references to its predecessors and successors in the pipeline. It references all state changes inside the module, allowing to collect statistics;
- State change connection: data transfer between two modules during the execution of a specific task. It contains links information about the difference in time between the end of execution of previous module and the start of the next one;
- Module connection: connectivity between two modules and corresponding state change connections (one for each task that passed through that module connection). Used to collect statistics over the entire execution.

The initial representation for the Smart Trace Collection is a list of state changes, from which all other entities can be extracted, through either visual interaction or Python scripting.

### 4.2 Smart Trace Composition and Exploration

We use the infrastructure offered by DEFOG to interact with the Smart Trace Collection. Once the Smart Trace Collection is imported into the application, the user can drag and drop its objects into the canvas, where they will receive visual representations. After being added to the scene, the system evaluates the data, and creates a default visualization composed of the graph representation of the dataflow, state change events as a Gantt chart, and tasks organized in a scatter plot.

The dataflow graph represents the average imbalance in workload among different inputs for each module by coloring each edge according to how late the data was delivered over the entire execution. Late data represents that a destination module stalls waiting for the data, and we color the edge accordingly (red represents high occurrence of late data). Modules are colored by their average run time. This default Gantt chart represents the full set of state change events collected during the execution, in the most basic way. Most of the future inspection will consist of duplicating, filtering and partitioning those events to show detailed execution for parts of interest.

The task scatter plot compares the time where its execution started, with the total idle time. The idle time is computed by the total amount of time where the data for this task is in the system but no execution is being performed. This visualization can show potential systematic problems within the execution scheduling policy of the system, if for example too many tasks have a high idle time; this means the corresponding data is being stored on buffers for a long time, just waiting for computational resources to become available.

### 5 RESULTS

We designed a set of experiments to validate the expressive power of the smart traces abstraction. We enumerate the goals of the trace analysis, followed by the results obtained for each dataflow construction. Before that, we give more details about the implementation.

#### 5.1 Dataflow System Implementation

Many of the existing dataflow systems are tightly coupled with larger specific environments (e.g. VisTrails, SCIRun) or have too rigid implementations (e.g. VTK, where the coordination of execution is not very centralized in a single entity.). For that reason we decided to implement our own minimalistic dataflow system in C++. Each module is implemented by deriving a base class. Besides describing input and output ports, the module can also specify both CPU and GPU implementations (the later using CUDA). The scheduler (which runs in its own thread) uses a single global priority queue for the execution requests. The priority is based on the order the corresponding tasks were submitted to the system (the first task submitted has the highest priority, so any execution corresponding to that task will be on the front of the queue); whenever an execution thread is available the scheduler looks for an execution request in the queue that can be executed, and assigns it to the thread. This constitutes a best-effort priority-based scheduling, as the order of completion of the tasks is not rigidly enforced. The centralized scheduler is the single point requiring instrumentation to collect the traces of the execution.

#### 5.2 Trace Analysis Goals

We designed our experiments to answer the following questions:

- **Q1** Identify the most time consuming modules in a dataflow and inspect its effect in the pipeline performance;
- **Q2** Identify which input tasks cause certain modules to take more time to compute;
- **Q3** Recognize which modules have their execution stalled because of other modules. This is relevant when a module has multiple inputs, and does not have all the inputs available to start executing; the available inputs have to stay in memory;
- **Q4** Visually inspect performance aspects of a dataflow running in a heterogeneous system composed of CPUs and GPUs;
- **Q5** Recognise runtime characteristics that are emergent from various implementation details.

Different dataflow constructions are used to evaluate the questions above. The first two dataflow constructions were synthetically generated to simulate one or more abnormal behaviors, which are supposed to be spotted upon analysis. Each synthetic dataflow is specified by its topology, the estimated run time costs of each module, and the number of tasks to be processed during execution. For example, we created a single module bottleneck situation, when a single module runs slower than the rest. This construction addresses Q1. Another example is to create an asymmetric workload, with multiple execution paths converging to a single join module at different speeds. In such situations, the join module has to wait for all the inputs to be available. Such construction addresses Q2. Finally, we can create intermittent slowdowns, where multiple inputs are processed, but a few of them will make some modules run slower (e.g. an image processing module that is optimized for power-of-two sized images.) This addresses Q3. To address Q4 we used an image processing pipeline with modules that can be executed both on CPU and GPU. Finally, Q5 was addressed by analyzing larger sets of inputs or with larger degrees of parallelism.

#### 5.3 Complex Pipeline

The first dataflow pipeline, shown on Figure 3 (left), is called Complex and contains multiple paths with different runtime costs joining at three points. The module Join1 runs much slower than the others. It is not only a bottleneck, but almost always guarantees that the input
for the following module (Join2) arrives late. Figure 3 (right) shows the result of an execution with 8192 inputs running with 256 worker threads on a 264-core SMP machine, with a time-compressed stacked line graph. The time scale was compressed non-linearly so each event in the graph has unit length, no matter how short or how long that event lasted. This shows more clearly short-lived behaviors without risking other events overlapping the same area of the chart. We can also derive from the figure that the scheduler favors a grouping of executions by module, with a small level of concurrency for most of the execution. As soon as data is available from the DataSrc module, it becomes possible to start the execution of Module0 and Module3, but there is little overlap between the former and the last two. A few snapshots of the active states at six instants during the execution is displayed below, coloring only the modules that are active at that point. This suggests that the synchronization and communication between the scheduler and the worker threads is comparable to the runtime of each execution, thus the scheduler is not able to keep the worker threads busy.

The same execution is visualized on Figure 4, with a full Gantt chart on top, and a single dataflow task (corresponding to the the 1811th input) displayed below. The network below is composed of the actual state change events for that task, copied directly from the Gantt chart visualization, and inter-connected through the dataflow topology. The “duration” property of the event objects was used as the label of each event, and connections were created between them and the Gantt chart to show where in time they occur. We can verify that module Join1 is really taking much longer to execute, but for this particular input it was also scheduled to executed much later in time than one would expect (e.g. after Module8’, colored dark blue.)

### 5.4 Irregular Pipeline

A second dataflow, called *Irregular* (Figure 5 (left)), has a module C having an arbitrarily irregular runtime: once every 17 inputs it runs much slower. This is characteristic that can be hard to identify by just looking at average run times. This dataflow was executed with 1024 inputs, with 4 working threads. Inside each pipeline module we have a scatter plot showing how long each task processed in that module lasts. A single execution, that was an outlier on module Src, is connected to executions in the other plots corresponding to the same dataflow task. It was also an outlier on modules A, D and Sink. Figure 5 (right) shows enlarged versions of the scatter plots of modules Src, C and Sink. The additional delay for slow executions on C is deterministic, and shows as a repeating pattern in the scatter plot.

The communication between modules represent the time since the data was available from the output port of a module until it was consumed by the next module. Delays in execution are shown in Figure 6 (left), with the start of the delay on the x-axis in seconds, and the duration of the delay in the y-axis, in nanoseconds, with a logarithmic scale. We can clearly see three groups of delays in the plot. The shortest delays (on the bottom of the chart) are towards the end, most likely because the scheduler queue gets smaller as the tasks complete; another group of delays vary more randomly, but on the same range, during the whole execution. And a few, but much longer delays, also seem to decrease in duration towards the end. On Figure 6 (right) we display two time annotated diagrams of the dataflow, showing on the left dataflow the average runtime and average delay for each module (inside module box) and connection (over each edge), and standard deviation for the same values on the right dataflow. All values use the same color map as the scatter plot, normalized between the minimum and maximum values for each quantity. This shows how both modules A and C are special (largest average and standard deviation run times, respectively). Delays before and after module A are much higher than the rest, and delay after module Src has the largest variation.

### 5.5 Edge Detection Pipeline

The last test case was an image processing application that performs edge detection on a large image composed of individual tiles (each tile being an independent input for the pipeline), composed into a single image output. Some stages of the pipeline (*Invert* and *Gaussian Blur* modules) have two implementations, one for the CPU and another for the GPU (using CUDA). While many embarrassingly parallel tasks
(such as many image processing operations) tend to perform better on the massively parallel hardware available on GPUs, it does not guarantee a performance boost; the overhead caused by data transfer and synchronization can easily offset the runtime trimmed from the execution if the implementation is not properly tuned.

The analysis is based on the execution with 512 input images with 3 MPixels each, running with 8 CPU worker threads and 3 GPU worker threads. Figure 1 shows an overview of the execution, with a Gantt chart of the entire runtime on top, and per-module Gantt charts, displayed inside the dataflow network, at different time windows. The gray rectangles at the top Gantt chart indicate the time windows corresponding to each network, and colors correspond to the module being executed. Figure 7 shows an analysis focused on the events in modules Invert and Gaussian Blur. A simple scatter plot for the durations of each execution shows mainly two groups of executions. The thread objects were added to the visualization and connected by lines to events they executed. The colors map to the thread related to the object, showing how the fastest executions were indeed performed by the GPU implementation of those threads. There is one outlier in the Gaussian Blur module that took considerably more time to complete.

The first 4 seconds of execution are shown on Figure 8. We observe at the top Gantt chart, which shows one line per thread, the first memory transfer to the GPUs (shown separately as a pseudo-module) taking a long time, due to the CUDA runtime initialization. Immediately after that the GPU threads stay idle for most of the time, which is expected as the Decode Image (blue) is too slow in comparison. Reorganizing the chart by module (middle) we can see an undesired behavior on the Decode Image module: it stops reading input images from the file system. By focusing on this module alone, we transform the chart into another visualization (bottom) with a stacked line graph, showing one thread per line, and notice a second undesired behavior: with too many thread trying to perform I/O at the same time the performance is likely to decrease, as the I/O buffers (and possibly the physical media) has to deal with many non-sequential accesses. An ideal execution for this module would have only a few threads active at time (to minimize concurrent I/O overhead), and no empty gaps, and a scheduling policy could provide this behavior.

6 Limitations
The design decision of using Python to represent and manipulate data implies an execution overhead inherent of the code being interpreted, with little to none optimization. This performance penalty is purely technological, and can be amortized if the chosen implementation uses a JIT interpreter. On the other hand, the dynamic aspect provides a convenient workflow without requiring the user to process data outside the environment to extract new information from the initial dataset.

The user interface is still very experimental, and very tied to programming concepts. For most of the exploration activity we feel it should provide enough functionality without requiring programming skills; on the other hand, it is reasonable to expect a user analysing this kind of data to be familiar with the basic programming concepts (see the supplementary text) required. We also want to be able to generate animations of trace data from within the platform. Some visualizations we generate require processing large amount of time-varying data, which are still challenging to produce real-time animations.

7 Conclusion and Future Work
In this work we presented different ways to look at parallel trace data, in particular to parallel dataflow traces. The motivation for this project came from our need to support debugging a parallel dataflow framework under development in our group. Current tools for trace analysis were limited to most of the time only displaying resource utilization Gantt charts. As we demonstrated along the work, trace data is rich and full of information that can be seen in different perspectives.

The notion of Smart Trace captures the idea of creating linked visualizations, which can be procedurally generated or simply generated from pre-recorded templates. The generality of the approach, enhanced with dataflow information, allowed us to observe this data in...
different ways, illustrate throughout the text with several examples.

In the future we would like to make the tool available to the public, but first we need to be able to perform code instrumentation for existing dataflow platforms, such as VTK. We believe it can be useful for the VTK user community, where we could test our system with diverse usage scenarios. This certainly would require to validate our technique with larger dataflow topologies, and we are currently exploring multi-resolution visualizations.

**Fig. 5.** (left) Each module of the Irregular pipeline displays a scatter plot of all executions, comparing the start time (x-axis, in seconds) with the duration (y-axis, in milliseconds). The gray arrows show the pipeline connections, while the black arrows connect executions related to one specific input. (right) Modules Src, C and Sink are shown zoomed in.

**Fig. 6.** Analysis of run times and delays in the Irregular pipeline. (left) Scatter plot of delays comparing start time (x-axis, in seconds) and duration (y-axis, in nanoseconds, log scale), with the color map used on its right. (right) Time annotated diagrams showing both averages and standard deviations of run times (modules) and delays (connections). This salients modules and connections with different behaviors, such as modules Src (largest average run time) and C (largest standard deviation run time), and connections around module A.

**Fig. 7.** Scatter plot for two modules (Invert and Gaussian Blur) showing duration of executions (y-axis) over time (x-axis) Lines connect the execution to the thread where it occurred. Observe that a single GPU execution in the Gaussian Blur module takes much longer to run.

**REFERENCES**


Fig. 8. Top and middle: two Gantt charts visualizations of two Smart Traces, one parametrized by thread and one by pipeline module; the time interval displayed is for the first 4 seconds of execution. The orange represents data upload to the GPU, and its first execution in each GPU thread (top three lines of the top chart) take longer to complete because of the CUDA runtime initialization. The bottom chart is a stacked line chart by interval displayed is for the first 4 seconds of execution.