

Tools for GPU Computing – Debugging and Performance Analysis of Heterogenous HPC Applications

Michael Knobloch¹, Bernd Mohr¹

© The Authors 2020. This paper is published with open access at SuperFri.org

General purpose GPUs are now ubiquitous in high-end supercomputing. All but one (the Japanese Fugaku system, which is based on ARM processors) of the announced (pre-)exascale systems contain vast amounts of GPUs that deliver the majority of the performance of these systems. Thus, GPU programming will be a necessity for application developers using high-end HPC systems. However, programming GPUs efficiently is an even more daunting task than traditional HPC application development. This becomes even more apparent for large-scale systems containing thousands of GPUs. Orchestrating all the resources of such a system imposes a tremendous challenge to developers. Luckily a rich ecosystem of tools exist to assist developers in every development step of a GPU application at all scales.

In this paper we present an overview of these tools and discuss their capabilities. We start with an overview of different GPU programming models, from low-level with CUDA over pragmatized models like OpenACC to high-level approaches like Kokkos. We discuss their respective tool interfaces as the main method for tools to obtain information on the execution of a kernel on the GPU. The main focus of this paper is on two classes of tools, debuggers and performance analysis tools. Debuggers help the developer to identify problems both on the CPU and GPU side as well as in the interplay of both. Once the application runs correctly, performance analysis tools can be used to pinpoint bottlenecks in the execution of the code and help to increase the overall performance.

Keywords: performance analysis, debugging, GPU computing.

Introduction

General purpose GPUs are now ubiquitous in high-end supercomputing. With the rise of deep learning and the convergence of simulation-based HPC and AI, GPU computing took a major leap forward. All but one (the Japanese Fugaku system, which is based solely on ARM processors) of the announced (pre-)exascale systems contain vast amounts of GPUs that deliver the majority of the performance of these systems. Thus, GPU programming will be a necessity for application developers using high-end HPC systems. However, programming GPUs efficiently is an even more daunting task than traditional HPC application development. This becomes even more apparent for large-scale systems containing thousands of GPUs. Orchestrating all the resources of such a system imposes a tremendous challenge to developers. Besides GPUs other accelerators have been tried, the most prominent being Intels Xeon Phi as a many-core architecture and FPGAs. However, the Xeon Phi has been discontinued and FPGAs are only a niche solution for very specific workloads or research projects, but not (yet) ready for production use in general HPC.

NVIDIA GPUs power most of today's GPU-enabled supercomputers. 136 systems in the TOP500 list of November 2019² are equipped with NVIDIA GPUs, including the number one and two systems, the U.S.-based Summit [33] and Sierra [25] supercomputers. Thus, we put a strong focus on NVIDIA GPUs in this paper.

Tools have always been an integral part of the HPC software stack. Debuggers and correctness checker help application developers to write bug-free and efficient code. Code efficiency can

¹Forschungszentrum Jülich GmbH, Jülich Supercomputing Center, Jülich, Germany

²<https://www.top500.org/list/2019/11/>

be improved by pinpointing bottlenecks with performance analysis tools. The tools community is working hard to provide tools that master the complexity of modern HPC systems [30], facing the same challenges when scaling up as the application developers themselves. Today, a rich ecosystem of tools exist to assist developers in every development step of a GPU application at all scales, from a workstation to a supercomputer.

In this paper we present an overview of these tools and discuss their capabilities. We present the currently dominant programming models for GPU computing and discuss their tool interfaces as the main method for tools to obtain information on the execution of a kernel on the GPU in section 1. Then we look into debuggers in section 2, which help to develop correct heterogenous applications that scale to several hundred or thousand of GPUs. Performance analysis tools, which help to use these resources efficiently, are discussed in section 3. Finally we conclude the paper and give an outlook on future developments in heterogenous supercomputing.

1. GPU Programming Models

For decades two programming paradigms dominated the HPC landscape – distributed-memory programming (inter-node) and shared-memory programming (intra-node). The main programming model for distributed-memory programming is MPI, the Message Passing Interface [28], which is used in virtually all HPC applications. MPI is a rather low-level interface, i.e. the user has to explicitly express the communication pattern and data transfers. Shared memory programming is mostly done via OpenMP [36], a directive-based API. For both MPI and OpenMP alternatives exist, like the PGAS (Partitioned Global Address Space) model for distributed memory or pthreads and TBB (Threading Building Blocks) for shared memory, but none come close in popularity to MPI and OpenMP.

With the advent of general purpose GPUs, things changed significantly. Now a new very powerful but also very complex architecture was thrown into the mix, yet on the other hand the old programming paradigms are still valid in order to create scaling HPC applications. There exist several programming models for GPUs, some are low-level like MPI, others are pragma-based like OpenMP. Some support only certain languages or specific vendor architectures, others are more open. So it is a challenge for an application developer to choose the right programming model for his application, but also for tools developers to choose which models to support. In this section we present various GPU programming models that suits different needs, CUDA and OpenCL as high-performance low-level interfaces, OpenACC and OpenMP as easy-to-use yet efficient directive-based approaches and KOKKOS and RAJA that aim for performance portability on a wide range of architectures. Where applicable we also give an introduction to the respective tools interface as the main source for tools to get information on the kernels running on the accelerator and the data transfers to and from the device.

1.1. CUDA

CUDA [32] is a parallel computing platform and programming model developed by NVIDIA for general computing on NVIDIA GPUs. It is a very low-level interface, i.e. the programmer has to specify every data movement and kernel launch explicitly. Given access to all hardware features of modern GPUs like Unified Memory, CUDA can yield the highest performance achievable on GPUs. However, this comes at the cost of a rather high development effort and non-portability. A rich set of libraries, both from NVIDIA directly and from third parties, are available for

CUDA, enabling developers to harness the power of CUDA without the need to deal with all the low-level details of the architecture. So far CUDA is the most popular programming model for GPU programming, thus most tools support CUDA to some extent. While CUDA itself is C++, CUDA bindings exist for many programming languages like C, Fortran (currently only for PGI compilers), Python and MATLAB.

1.1.1. CUPTI – The CUDA Performance Tools Interface

The NVIDIA CUDA Profiling Tools Interface (CUPTI) provides performance analysis tools with detailed information about how applications are using the GPUs in a system. CUPTI provides two simple yet powerful mechanisms that allow performance analysis tools to understand the inner workings of an application and deliver valuable insights to developers. The first mechanism is a callback API that allows tools to inject analysis code into the entry and exit point of each CUDA C Runtime (CUDA Runtime) and CUDA Driver API function. Using this callback API, tools can monitor an applications interactions with the CUDA Runtime and driver. The second mechanism allows performance analysis tools to query and configure hardware event counters designed into the GPU and software event counters in the CUDA driver. These event counters record activity such as instruction counts, memory transactions, cache hits/misses, divergent branches, and more. This enables automated bottleneck identification based on metrics such as instruction throughput, memory throughput, and more.

1.2. OpenCL, SYCL and oneAPI

The aim of OpenCL, the *Open Computing Language*, is to provide a vendor independent programming interface for all kinds of computing devices, from CPUs over GPUs to FPGAs. OpenCL is developed by the Khronos Group, an open industry consortium of over 100 leading hardware and software companies. OpenCL, like CUDA, is a low-level API where the kernels are written in the OpenCL C++ kernel language, a static subset of C++14.

To ease the development of heterogenous applications, the Khronos group developed SYCL as an abstraction layer build on the concepts, portability and efficiency of OpenCL. SYCL allows the developer to program on a higher level than OpenCL, while still having access to lower-level code. A lot of the boilerplate code of OpenCL is removed by SYCL and a single-source programming, where host and device code are contained in the same source file, is enabled.

The newest member in the OpenCL language space is Intel's oneAPI with DPC++ (*Data Parallel C++*), which in turn is built upon SYCL. Due to its recent Beta release and the – at the time of writing – limited availability of hardware, the support of tools for oneAPI could not be evaluated for this paper. However, it is clear that the well-known Intel tools VTune and Advisor will have rich support for oneAPI. The most interesting and unique feature of the Intel Advisor will be an analysis of the potential gain of offloading a sequential code path to an accelerator.

It will be interesting to see how oneAPI will be adopted by the HPC community and how the tools support for SYCL and oneAPI develops. Codeplay, a compiler vendor and active part in the SYCL community, recently announced SYCL support for NVIDIA GPUs [38], which could dramatically increase the interest in SYCL as a portable API as it significantly increases to potential user-base.

1.2.1. The OpenCL Profiling Interface

OpenCL provides a very basic interface to get profiling information on memory operations and kernel launches. If profiling is enabled, the function `clGetEventProfilingInfo` returns timing information of OpenCL functions that are enqueued as commands to a command-queue. The most interesting for performance analysis are the begin and end timestamps of kernel launches. The SYCL specification defines a similar profiling interface. However, most tools with OpenCL support use some form of library wrapping to obtain information of the OpenCL execution.

1.3. OpenACC

The OpenACC (Open ACcelerator) API [34] describes a collection of compiler directives to specify loops and regions of code to be executed in parallel on a multicore CPU, or to be offloaded and executed in parallel on an attached accelerator device, providing portability across operating systems, CPUs, and accelerators. With directives for C/C++ and Fortran, OpenACC covers the most important programming languages for HPC.

OpenACC eases the development of heterogenous applications as it relieves the user from explicit accelerator and data management as well as data transfers to and from the device. Data management is handled with the *data* construct, where *enter data* and *exit data* directives can be used to control data transfers between host and device. Two fundamental compute constructs, *kernels* and *parallel* can be used to offload the execution of code blocks to an accelerator.

While OpenMP is a prescriptive programming model, i.e. the developer explicitly states *how* to split the execution of loops, code regions and tasks among well-defined teams of threads, OpenACC is more descriptive model, telling the compiler *where* it is safe to parallelize loops or offload kernels and *what* data has to be transferred. This enables the compiler to perform more optimizations and generate faster code [43].

1.3.1. OpenACC Profiling Interface

OpenACC provides a profiling interface for both profile and trace data collection. This interface provides callbacks that are triggered during runtime if specific events occur. Three types of events are supported: data events, launch events and other events. Data events cover the allocation/deallocation of memory on the accelerator as well as data transfers. Launch events trigger before and after a kernel launch operation. Other events include device initialization and shutdown as well as wait operations [10]. However, these events only give host-side information. For information on the device the respective tools interface of the backend has to be used.

1.4. OpenMP

OpenMP is a directive-based API already well known for shared-memory parallelization on CPUs which is easy to learn. It also offers a path to more portable GPU-accelerated applications. Like OpenACC, one of the goals of the OpenMP standard is to minimize the need for applications to contain vendor-specific statements. Thus, codes are portable across all supported GPU architectures.

Pragmas to offload work on general purpose GPUs have been introduced in OpenMP 4 [35], the OpenMP device constructs. The *target* construct is required to specify a region to be launched

on the device. *Target data* maps the variables on the device. The *teams* pragma inside target spawns the set of teams with multiple OpenMP threads. The *distribute* construct partitions the iterations and maps it to each team.

1.4.1. The OpenMP Tools Interfaces

Unlike the other programming interfaces, OpenMP since version 5 [36] provides two tools interfaces, OMPT for performance analysis tools and OMPD for debuggers [12].

OMPT [13] is a portable interface for both sampling-based and instrumentation-based performance analysis tools. Like the other tool interfaces, OMPT provides callbacks for defined OpenMP events, like the begin of a parallel region or the start of a offloaded kernel. It also maintains the tools data for OpenMP scopes and it provides signal-safe inquiry functions to get OpenMP runtime information. OMPT is intended for first-party tools, i.e. tools that are linked into or loaded from the OpenMP application.

OMPD, the OpenMP debugging interface, on the other hand is an interface for third-party tools, i.e. tools that live in a different process from the OpenMP application. This interface allows external tools to inspect the OpenMP state of a running program via callbacks. The debugger has no direct access to the OpenMP runtime, it interacts with it through the OMPD architecture and the OMPD interface is transparent to the OpenMP application. The OMPD library can be used to debug a running program as well as core files generated when the application aborted due to an error.

1.5. Kokkos and RAJA

As stated above, HPC programming models didn't change for a long time, which gave application developers some confidence that their application will perform on the next generation of machines. With an increased variability in architectures and programming models that does not hold any more. An application tuned for a specific platform could perform badly on the next system, which could be completely different from the current one. Further, applications and libraries that are used universally need some kind of assurance to perform well on a wide range of architectures.

In the scope of the Exascale Computing Project [29], two projects emerged that strive for performance portability by providing an abstraction layer over the existing programming models. Both originate from US national laboratories, one is Kokkos [11], developed at Sandia, and the other one RAJA [2] from LLNL. The abstraction layers include memory and execution spaces, data layout (i.e. the data layout might change depending on the architecture the application is compiled for) and parallel execution.

Both Kokkos and RAJA currently provide only C++ interfaces and only have a CUDA backend for offloading work to a GPU, though support for other programming models is likely to follow.

1.5.1. The Kokkos Profiling Interface

Kokkos provides a set of hooks for profiling libraries to interface with the Kokkos runtime [19]. These hooks can be implemented in the form of callbacks within a shared library. Upon start of the application, the Kokkos runtime loads the library, checks for implemented callbacks, and invokes the performance monitor via corresponding callbacks. Currently Kokkos

supports callbacks for initialization and finalization of the runtime, deep data copies, and the three parallel execution models `parallel_for`, `parallel_reduce`, and `parallel_scan`. Similar to the OpenACC profiling interface only events on the host are triggered, though device events can be captured with CUPTI. RAJA unfortunately does not provide a profiling interface at this time.

2. Debuggers

Developing correct parallel programs is already a daunting task, adding the complexity of GPUs to the mix makes that endeavour even harder. This holds especially when using low-level programming paradigms, where the user is responsible for correct memory management and data movement. Luckily, several debugging solutions exist to assist the application developer in finding and fixing bugs, both at small and large scale.

Table 1. Debugger compatibility matrix showing the level of support of different debuggers for the most popular GPU programming models [Status Feb. 2020]

Tool	CUDA	OpenACC	OMPD	OpenCL
CUDA-MEMCHECK	yes	partly (CUDA kernels)	no	no
CUDA-GDB	yes	partly (CUDA kernels)	no	no
TotalView	yes	yes	prototype	no
DDT	yes	yes	no	no

Debuggers, especially those for large-scale HPC systems are very complex and sophisticated pieces of software and virtually no open source solution exist here. The main debugging solutions for GPU programming right now are the NVIDIA provided CUDA-MEMCHECK and CUDA-GDB and the commercially available TotalView and DDT. Table 1 shows the supported GPU programming models of each of these debuggers. There is very good support for CUDA and OpenACC (where the NVIDIA tools support the debugging of the generated CUDA kernels), but nearly no support for the other programming models. TotalView showed a prototype with support for OpenMP offloading using an experimental OMPD-enabled OpenMP runtime. There exist a couple of debuggers for OpenCL, but none proved usable for complex HPC applications³.

2.1. NVIDIA Debugging Solutions

NVIDIA realized the importance of debugging for novel programming paradigms right from the beginning and shipped debugging tools right from the beginning with the CUDA toolkit [17]. These tools can be used standalone from the command-line, but are also integrated in the Nsight IDE [20], NVIDIA's development platform for CUDA and OpenACC applications. An example debugging session is shown in Fig. 1.

2.1.1. CUDA-MEMCHECK

CUDA-MEMCHECK is like valgrind for GPUs, a very powerful memory tracker and analysis tool. Hundreds of thousands of threads running concurrently on each GPU can be monitored.

³Some promising OpenCL debuggers are usable only on Microsoft Windows, which is not the intended platform for HPC applications

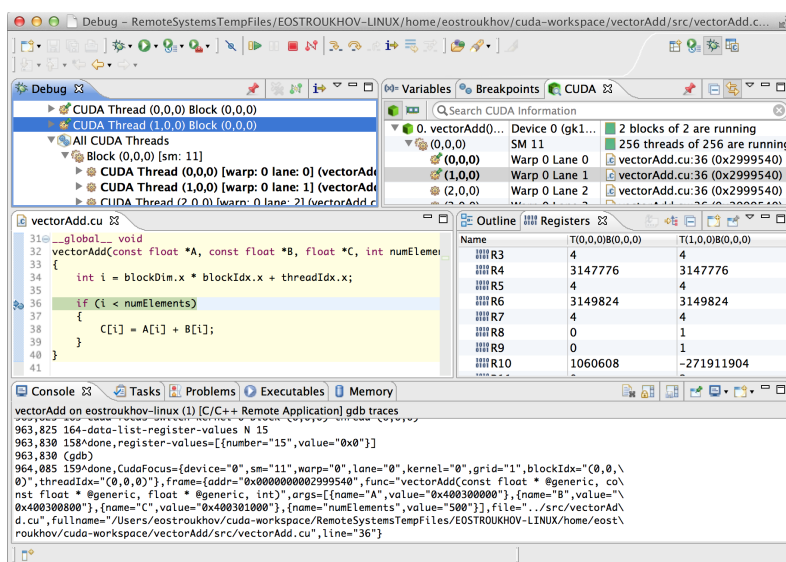


Figure 1. Debugging a CUDA application from within the Nsight IDE

It reports detailed information about global, local, and shared memory access errors (e.g. index out-of-bounds or misaligned memory accesses) and runtime executions errors (e.g. stack overflows and illegal instructions). Potential race conditions can also be detected with CUDA-MEMCHECK. In case of an error, CUDA-MEMCHECK displays stack back-traces on host and device.

2.1.2. CUDA-GDB

CUDA-GDB is, as the name indicates, an extension to gdb, *the* Unix debugger. Simultaneous debugging on the CPU and multiple GPUs is possible. The user can set conditional breakpoints or break automatically on every kernel launch. It is possible to examine variables, read/write memory and registers and inspect the GPU state when the application is suspended. Memory access violations can be analyzed by running CUDA-MEMCHECK in an integrated mode to detect the precise causes.

2.2. TotalView

TotalView⁴ is a symbolic debugger specifically designed for HPC applications written in C/C++, Fortran or Python. Noteworthy are the analysis capabilities for heavily templated C++ codes with complex data types. Advanced Memory Debugging allows to keep track of all memory accesses and memory allocations and deallocations to find memory leaks and corrupted memory. Another feature that sets TotalView apart from the competition is reverse debugging, i.e. the program execution is recorded and the user can step back from the point where the error occurred. This is especially helpful in fixing non-deterministic bugs. TotalView features full control over processes and threads with the ability to stop and debug an individual thread or groups of threads or processes. Debugging of CUDA [18] and OpenACC applications is supported with the possibility to debug multiple GPUs on a single node or multiple nodes across a cluster. Here it is possible to seamlessly set breakpoints in host and device code.

⁴<https://totalview.io/products/totalview>

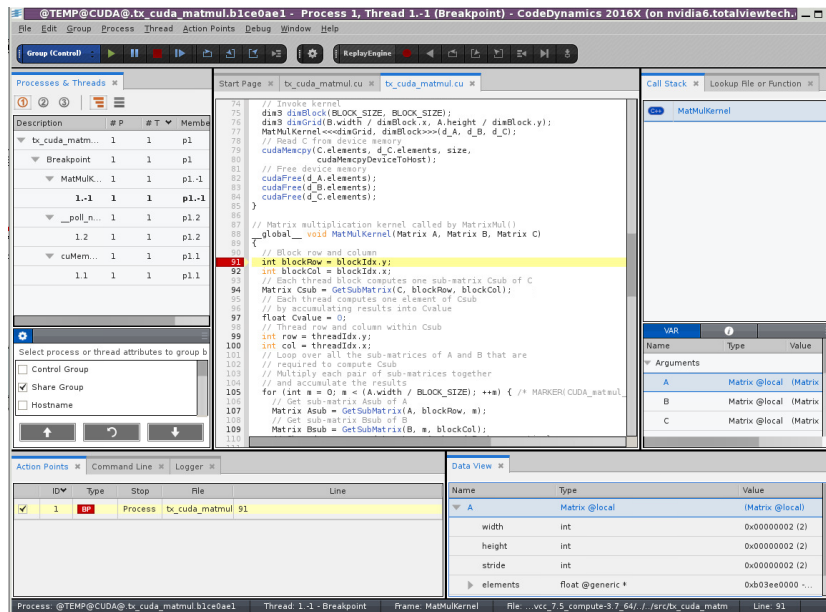


Figure 2. Debugging a matrix-multiplication kernel with TotalView

Figure 2 shows a screenshot of a CUDA debugging session using the new TotalView GUI, which greatly improves the usability.

2.3. Arm DDT

DDT⁵ is another commercial debugger with a modern interface and very similar features to TotalView. It supports all major HPC programming languages with a special focus on complex C++ applications. Multi-process and multi-thread support is a matter of course. DDT also features advanced memory debugging and visualizations of huge data sets. Like TotalView, DDT supports debugging of CUDA and OpenACC applications with a fine-grained thread control, as shown in Fig. 3. DDT is available standalone or together with the Arm profiling tools in the Arm Forge suite⁶.

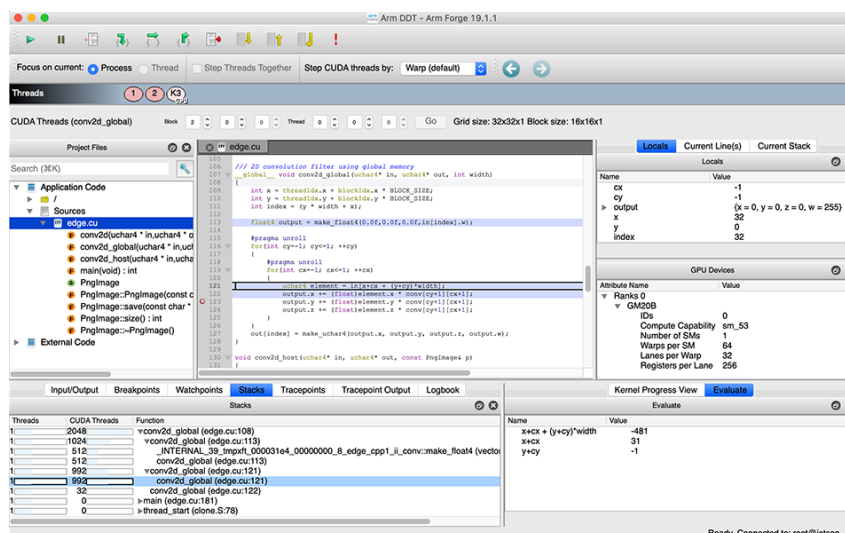


Figure 3. Arm DDT

⁵<https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt>

⁶<https://www.arm.com/products/development-tools/server-and-hpc/forge>

3. Performance Analysis Tools

Performance analysis tools are an integral component in the HPC software stack for decades and many application developers were exposed to profilers to a certain degree. There are many tools for all kinds of analyzers, some are vendor provided and thus tied to a specific platform, some are commercial and several open source. The latter are usually developed at universities or national research laboratories with larger supercomputers. The tools community, which has a long history of collaboration, started adding GPU support relatively early [26], though the programming models and amount of features supported varies significantly between tools.

Though we commonly refer to performance analysis tools as profilers, we distinguish between trace-based tools, which store all events with timestamps and profile-based tools, which only store statistical information like the number of calls to a specific routine and the total time spend in that routine. Several tools can generate both profiles and traces and are thus universally applicable.

Table 2. Performance tool compatibility matrix showing the support for GPU programming models of several popular performance analysis tools [Status Feb. 2020]

Tool	CUDA	OpenACC	OMPT	OpenCL
NVIDIA Tools	yes	yes	no	no
ARM Tools	yes	no	no	no
Score-P	yes	yes	prototype (no offload)	yes
TAU	yes	yes	prototype (no offload)	yes
HPCToolkit	yes	no	yes (experimental runtime)	no
Extrae/Paraver	yes	no	no	yes

Tool support for the various GPU programming models varies significantly. The tool compatibility matrix for some of the most popular and wide-spread performance analysis tools is shown in Tab. 2. CUDA is supported by all the tools we consider. This is partly because CUDA was the first programming model for GPUs, but also because NVIDIA provides a very powerful and easy to use tools interface with CUPTI. Half of the tools support OpenACC or OpenCL, so there are options for all application developers. Several tools are working on supporting OpenMP offload to GPUs, but there is currently no public OpenMP runtime that implements OMPT for target directives. However, both Score-P and TAU already support OMPT on the host-side. HPCToolkit showed a prototype with OpenMP offload support using an internal experimental OpenMP runtime that implements OMPT for target directives.

3.1. NVIDIA Tools

NVIDIA realized early on that good tools (and a good documentation) are a necessity for a new platform to gain traction. So NVIDIA began shipping their own profiler nvvp, the NVIDIA Visual Profiler, shortly after the release of CUDA. It is an integral feature of the CUDA tool-kit since then, so it is available on all CUDA-enabled platforms, without the need for a third-party tool. After several years, nvvp began to show scalability (and maintenance) issues and will be deprecated in a future CUDA release. Luckily, two new tools, Nsight Compute and Nsight System, are ready to fill that gap.

3.1.1. NVIDIA Visual Profiler

For many years, nvvp [5] was the de-facto standard profiler for CUDA applications. It presents a unified CPU and GPU timeline including CUDA API calls, memory transfers and kernel launches. For a more detailed analysis of CPU activities, users can annotate the source code using the NVIDIA Tools Extension (NVTX) [24]. It supports all the advanced features of recent CUDA versions like Unified Memory, with CPU and GPU page faults and data migrations shown in the timeline. Upon selection of a specific kernel, nvvp shows a detailed low-level kernel analysis with performance metrics collected directly from GPU hardware counters and software instrumentation. Nvvp can compare results across multiple sessions to verify improvements from tuning actions. Another unique feature is an Automated or Guided Application Analysis with graphical visualizations to help identifying optimization opportunities. The Guided Analysis provides a step-by-step analysis and optimization guidance. The Visual Profiler is available as

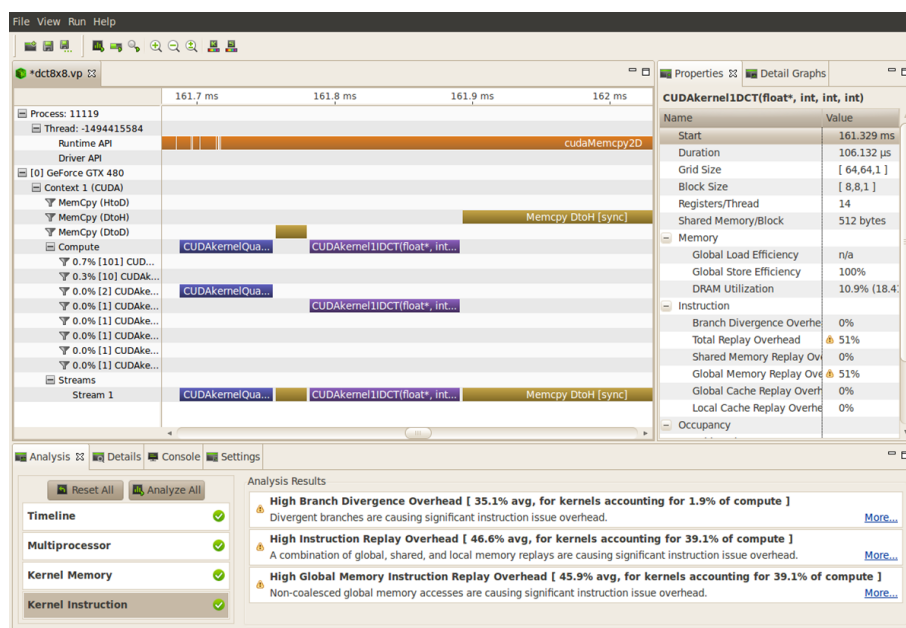


Figure 4. The NVIDIA Visual Profiler showing the timeline of the application execution, a detailed analysis of the selected kernel, and the results of the guided analysis

both a standalone application, as shown in Fig. 4 and, like NVIDIA's debugging solutions, as part of the Nsight IDE.

3.1.2. Nsight Compute

NVIDIA Nsight Compute⁷ is an interactive kernel profiler for CUDA applications. It provides similar features to nvvp's low-level kernel analysis, i.e. detailed performance metrics and the guided performance analysis. Nsight Compute provides a customizable and data-driven user interface (as shown in Fig. 5) Further, it has a command-line mode for manual and automated profiling and can be extended with analysis scripts for post-processing results. Additionally, its baseline feature allows users to compare results directly within the tool, very much like in the Visual Profiler.

⁷<https://developer.nvidia.com/nsight-compute-2019.5>

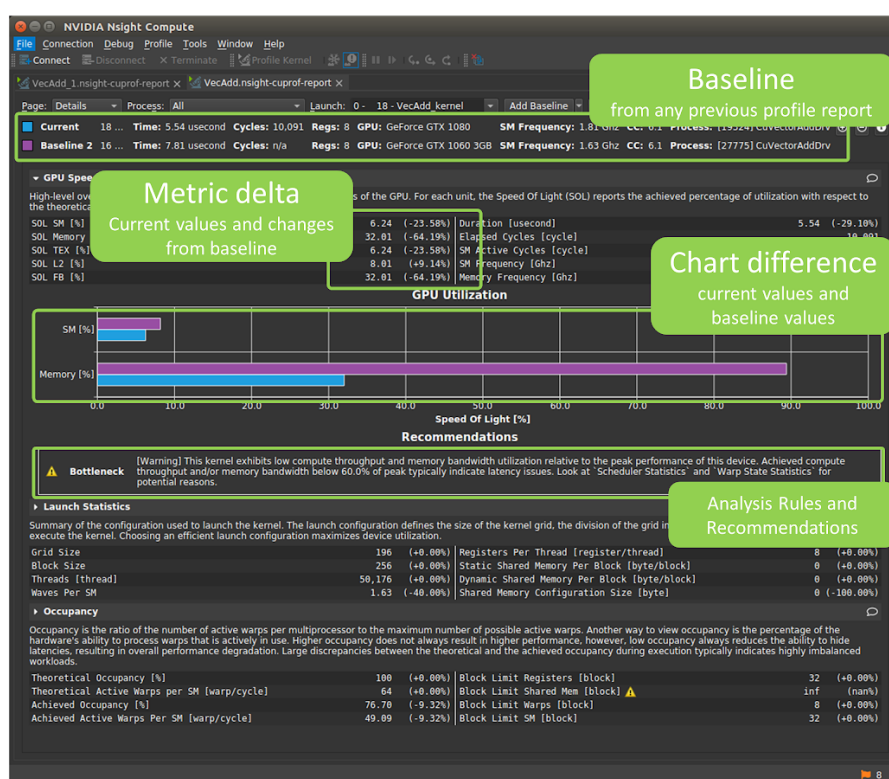


Figure 5. Nsight compute showing its detailed kernel analysis with the baseline comparison

3.1.3. Nsight Systems

NVIDIA Nsight Systems⁸ is a system-wide timeline-based performance analysis tool. It is designed to visualize the complete application execution help to identify the largest opportunities to optimize, and tune to scale efficiently across any quantity or size of CPUs and GPUs. Users will be able to identify issues, such as GPU starvation, unnecessary GPU synchronization, and insufficient overlap with CPU computation. An example timeline is shown in Fig. 6. It is possible to zoom in to any level of detail. Kernels showing unexpected behavior can be analyzed in detail with Nsight Compute, launched directly from the Nsight Systems GUI. NVTX is supported to get a more detailed picture of the CPU utilization. Currently Nsight System is focused on a single process, with more advanced support for MPI and OpenMP planned for a future release.

3.2. ARM Tools

Arm, since the acquisition of Alinea in 2016, provides several commercial cross-platform performance analysis tools, that can be obtained standalone or together with DDT in the Arm Forge⁹ suite.

3.2.1. Performance Reports

Arm Performance Reports is a gateway to the world of performance analysis. It is a very low-overhead tool working on unmodified optimized binaries that generates a one-page report characterizing the application performance at a rather high level. Performance Reports analyzes

⁸<https://developer.nvidia.com/nsight-systems>

⁹<https://www.arm.com/products/development-tools/server-and-hpc/forge>

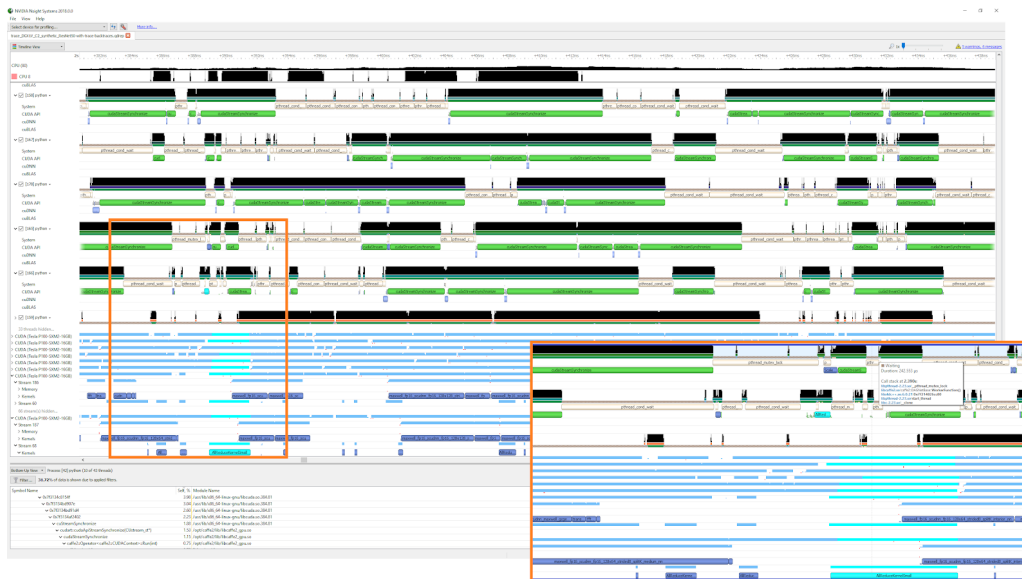


Figure 6. Nsight Systems showing the CPU timeline at the bottom and the activity of various CUDA stream on top

CPU utilization, MPI communication behavior, I/O and memory usage as well as accelerator usage. For all these categories it presents three to four sub-metrics to give more detailed information, e.g. the ratio of scalar and vector operations. For issues found, Performance Reports gives hints on how to proceed with more sophisticated analysis tools. An example of the accelerator breakdown of Performance Reports is shown in Fig. 7. This only gives a very brief overview of the GPU utilization, but in this case indicates that a thorough analysis with more advanced tools might be beneficial.

3.2.2. MAP

Arm MAP [21] is a fully featured cross-platform source level performance analysis tool. It supports low-overhead sampling-based profiling of parallel multi-threaded C/C++, Fortran and Python codes. MAP providing in-depth analysis and bottleneck pinpointing to the source line as well as an analysis of communication and workload imbalance issues for MPI and multi-process codes. For accelerators, MAP offers a detailed kernel analysis with data obtained via CUPTI. This includes a line-level breakdown of warp stalls. Possible reasons for warp stalls include execution and memory dependencies or barriers. Knowing the reason for warp stalls can help the developer tuning the code accordingly. However, MAP currently supports only kernels generated

Accelerators

A breakdown of how accelerators were used:

GPU utilization	47.8%	<div style="width: 47.8%; height: 10px; background-color: #666666;"></div>
Global memory accesses	1.6%	
Mean GPU memory usage	0.8%	
Peak GPU memory usage	0.8%	

GPU utilization is low; identify CPU bottlenecks with a profiler and offload them to the accelerator.

The peak GPU memory usage is low. It may be more efficient to offload a larger portion of the dataset to each device.

Figure 7. Performance Reports accelerator breakdown

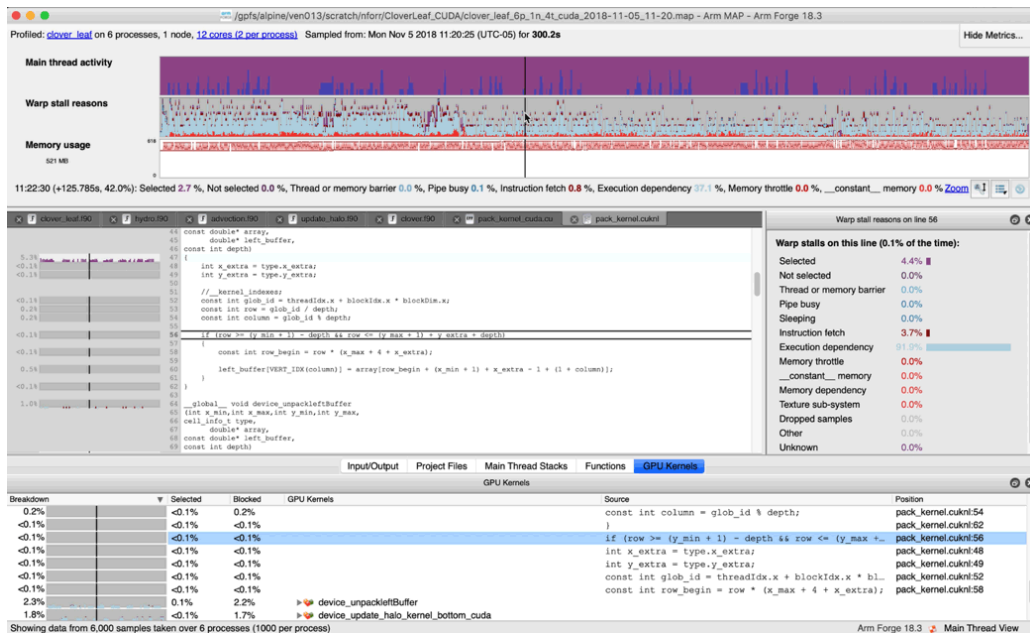


Figure 8. Arm MAP screenshot of a CUDA application analysis. It features the detailed warp stall analysis next to the source code

by CUDA-C++, not those generated by OpenACC, CUDA-Fortran or OpenMP offload. Figure 8 shows an example of MAP analyzing a CUDA application.

3.3. The Score-P Ecosystem

Score-P [23] is a community instrumentation and measurement infrastructure developed by a consortium of performance tool groups. It is the next-generation measurement system of several tools, including Vampir [22], Scalasca [16], TAU [42] and Periscope [3]. Common data formats for profiling (CUBE4) and tracing (OTF2 [14]) enable tools interoperability. Figure 9 gives an overview of the Score-P ecosystem. On the bottom are the various supported programming paradigms, which are implemented as independent adapters interacting with the measurement

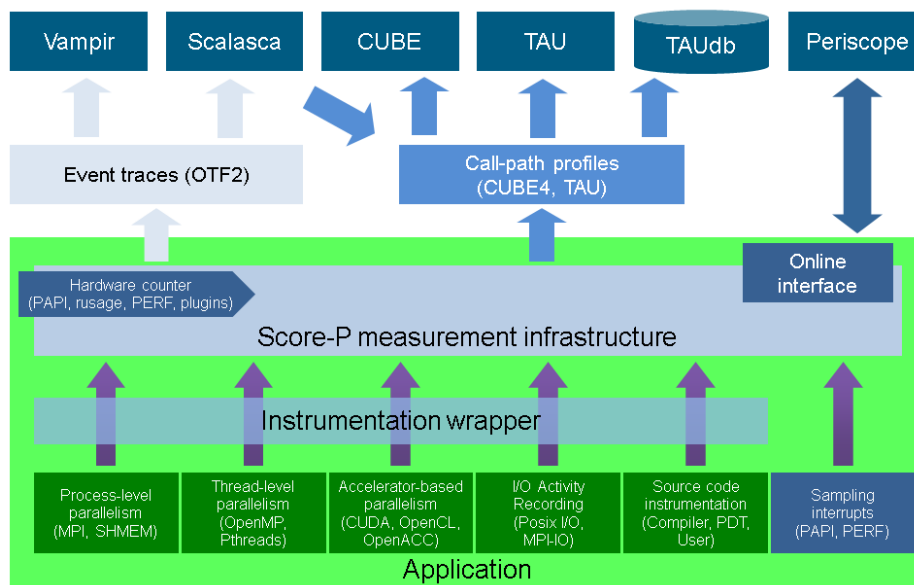
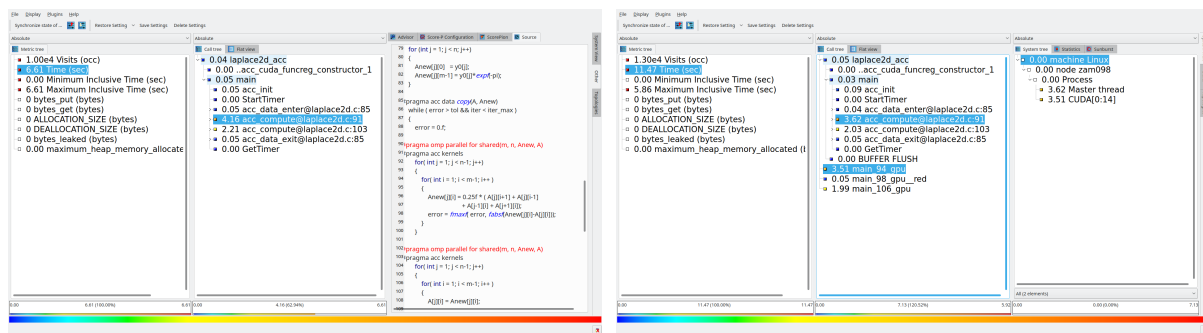


Figure 9. The Score-P ecosystem

system core. That eases adding support for new paradigms. The measurement data can be enriched with hardware counter information from PAPI [31], perf, or rusage. Score-P supports all major GPU programming models with CUDA [26], OpenACC [8], and OpenCL [9]. OMPT support for host-side measurement was recently added [15] and there is ongoing work to support OpenMP target directives [7]. Score-P also features a sampling mode for low-overhead measurements. It supports both profiling and tracing for all adapters. Profiles are generated in the CUBE4 format, that can be analyzed by TAU or Cube [39].

Cube is the performance report explorer for Score-P profiles as well as for the Scalasca trace analysis. The CUBE data model consists of a three-dimensional performance space with the dimensions (i) performance metric, (ii) call-path, and (iii) system location. Each dimension is represented in the GUI as a tree and shown in one of three coupled tree browsers, i.e. upon selection of one tree item the other trees are updated. Non-leaf nodes in each tree can be collapsed or expanded to achieve the desired level of granularity. Figure 10 shows a profile of a simple OpenACC application Cube GUI. On the left (Fig. 10a), the results of a pure OpenACC measurement are shown. Due to restrictions of the OpenACC tools interface, only the host-side calls are visible. However, if Score-Ps CUDA support is enabled as well, also the kernels generated by OpenACC get recorded (Fig. 10b).



(a) OpenACC only

(b) OpenACC + CUDA

Figure 10. Screenshots of Cube showing the Score-P measurement of a simple OpenACC application, (a) with only OpenACC enabled, showing only the host side, and (b) with OpenACC and CUDA enabled, which additionally shows the kernels and device activity

OTF2 traces generated by Score-P can be analyzed automatically with Scalasca, which determines patterns indicating performance bottlenecks, and manually with Vampir. Unfortunately, Scalasca currently does not support the analysis of traces containing GPU locations, but can be used to analyze the communication of multi-node heterogenous programs if the corresponding adapter for the GPU programming model is disabled, i.e. only host-side events are recorded. In contrast to traditional profile viewers, which only present aggregated values of performance metrics, Vampir allows the investigation of the whole application flow. The main view is the Master Timeline which shows the program activity over time on all processes, threads, and accelerators. An example is shown in Fig. 11.

The Master Timeline is complemented by several other views, timelines, and tables, e.g. the Process Timeline to display the application call stack of a process over time or a Communication Matrix to analyze the communication between processes. Any counter metrics, e.g. from PAPI or counter plugins, can be analyzed across processes and time with either a timeline or as a heatmap in the Performance Radar. It is possible to zoom into any level of detail, all views are updated automatically to show the information from the selected part of the trace.

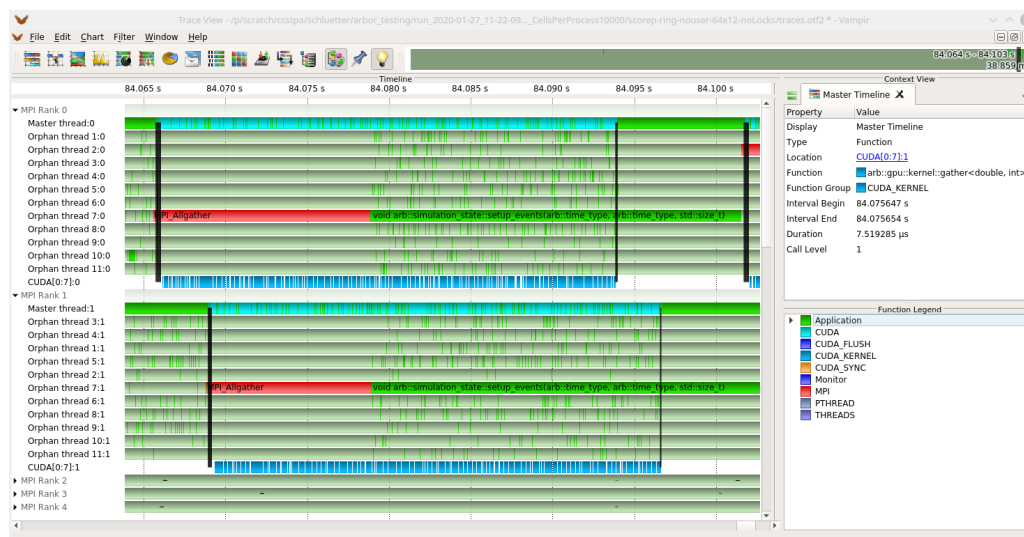


Figure 11. Vampir screenshot of an MPI + C++11 threads + CUDA application, showing a kernel launch on two processes in the Master Timeline. The line between the master process and the CUDA device indicates a data transfer, the thickness of the line represents the amount of data transferred

3.4. TAU

TAU [42] is a very portable tool-set for instrumentation, measurement and analysis of parallel multi-threaded applications. It features various profiling modes as well as tracing and various forms of code instrumentation as well as event-based sampling. All major HPC programming languages (C/C++, Fortran, Python) and programming models (MPI, OpenMP, Pthreads) are supported by TAU. TAU offers the widest support for accelerators, it allows measurement of CUDA [27], OpenACC, OpenCL, Kokkos [41], and also AMDs ROCm+HIP. For the analysis of 3-dimensional profile data, TAU includes ParaProf, which – like Cube – shows performance metric, call-path and location for an easy and quick investigation of bottlenecks. Figure 12 shows the visualization of a TAU trace file with Jumpshot¹⁰.

3.5. Extrae/Paraver

Extrae¹¹ is a measurement system to generate Paraver trace files for post-mortem analysis. It supports C/C++, Fortran, and Python programs on all major HPC platforms, i.e. Intels x86, NVIDIA GPUs, Arm, and openPOWER. Extrae features several measurement techniques, which are configured through an XML file. The main source of information in Extrae is preloading shared libraries that substitutes symbols for many parallel runtimes, e.g. MPI, OpenMP and CUDA. Extrae also support dynamic instrumentation by modification of the application binary and parallel runtimes via Dyninst [4]. Further, Extrae supports sampling via signal timers and hardware performance counters. Since the Paraver trace format has no predefined semantics, adding support for new paradigms is relatively straightforward.

Paraver [37, 40] is a very flexible data browser working on the trace files generated by Extrae. Flexible means that there is no fixed set of metrics, the metrics can be programmed in the tool itself. Paraver offers a large selection of views, e.g. timelines, histograms, and tables, that can

¹⁰<https://www.mcs.anl.gov/research/projects/perfvis/software/viewers/index.htm>

¹¹<https://tools.bsc.es/extrae>

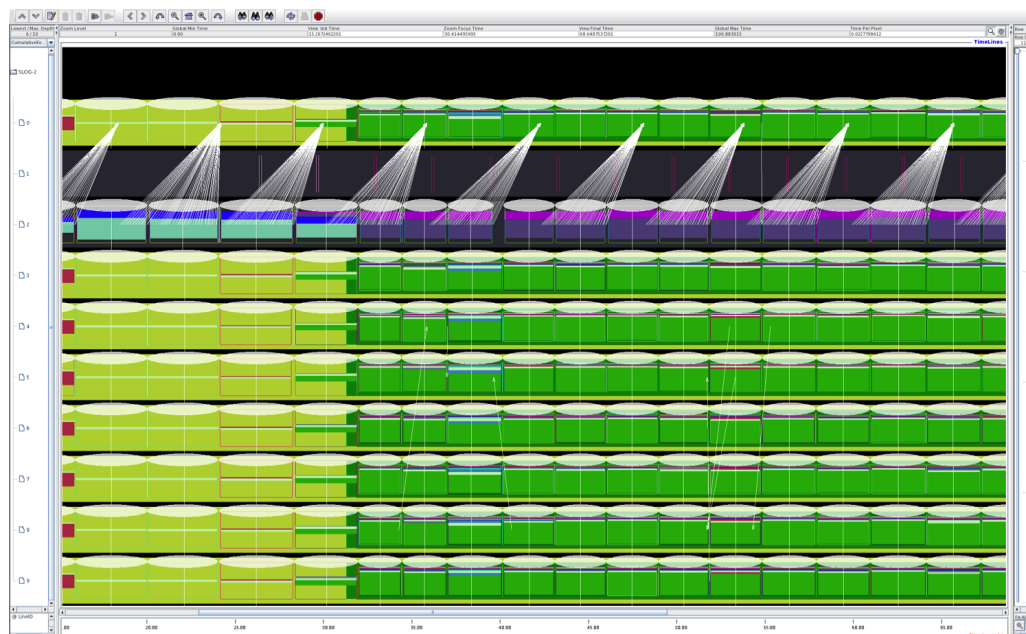


Figure 12. Jumpshot screenshot of a TAU trace measurement of a CUDA application

be combined to show virtually all the information that is present in the data. A view (or a set of views) can be saved as a Paraver configuration and recalculated with another trace file. CUDA streams are displayed in Paraver like any other data source, e.g. MPI processes or OpenMP threads.

3.6. HPCToolkit

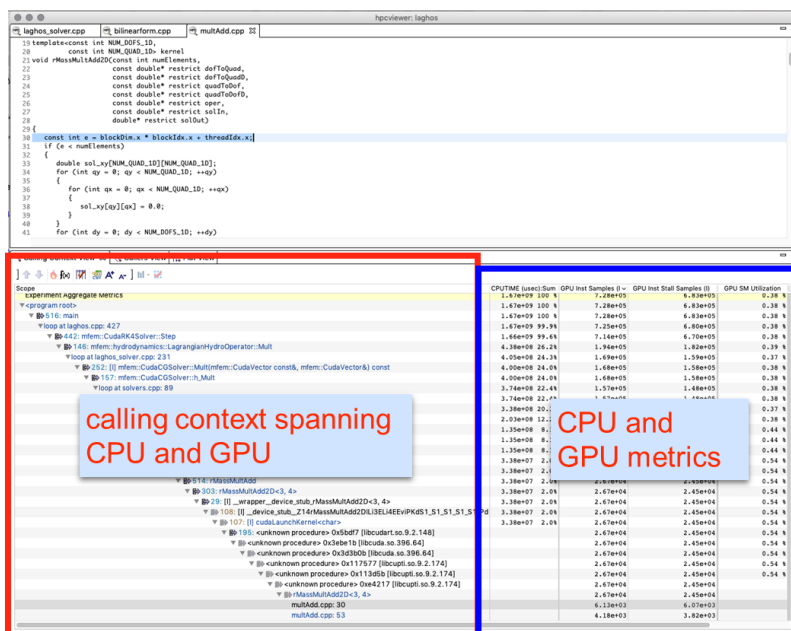


Figure 13. hpcviewer showing the profile of a CUDA application

HPCToolkit [1] is an integrated suite of tools for measurement and performance analysis of applications at all scales. Working on unmodified, fully optimized executables, it uses sampling to generate both call-path profiles and traces, independent of the language used. It supports multi-process (MPI) and multi-threaded (OpenMP, Pthreads) applications, but features no collection

or analysis of communication and I/O metrics. For GPU analysis, it supports CUDA [6] and, as a currently unique feature, OpenMP offload by shipping an experimental OpenMP runtime that implements OMPT for target constructs. It measures the execution time of each GPU kernel as well as explicit and implicit data movements. For CUDA codes it uses program counter sampling to pinpoint hotspots and to calculate the utilization of the GPU. A powerful analysis features of HPCToolkit is the blame shifting from symptoms to causes, so the user can quickly identify the real bottlenecks.

HPCToolkit consists of multiple programs that work together to generate a complete picture. **hpcrun** collects calling-context-sensitive performance data via sampling. A binary analysis to associate calling-context-sensitive measurements with source code structure is performed by **hpcstruct**. The performance data and the structure information are combined by **hpcprof** and finally visualized by **hpcviewer** for profiles and **hpctraceviewer** for traces.

Conclusion

In this paper we showed that tools can support developers in programming heterogeneous codes on current supercomputers, both in writing correct bug-free (debuggers) and efficient (performance analysis tools) applications.

There is one point in GPU programming where tools can't help – the decision which programming model to use. However, regardless of the choice, there is at least some tools support for each of the programming models. Due to the dominance of NVIDIA GPUs in today's data-centers, currently most developers choose CUDA or OpenACC, which are also the models with the best tools support. To use the tools as efficient as possible we recommend for that case to use the NVIDIA tools on a single node (with possibly multiple GPUs) when developing or porting the application to GPUs. When scaling up, i.e. inter-node data distribution and communication becomes an issue, we recommend the usage of more sophisticated tools like Score-P or TAU, which offer dedicated communication analysis. Errors occurring at scale can be debugged efficiently using TotalView or DDT.

Most supercomputing centers offer support to their users in porting and tuning applications to GPU architectures, sometimes in dedicated labs, like the JSC/NVIDIA Application Lab¹². The tools community is also actively supporting users via mailing lists and trainings, e.g. the VI-HPS Tuning Workshops¹³, which offer multi-day hands-on workshops covering a range of different tools.

So far the dominant player in GPU-enabled supercomputing is NVIDIA, but with the announced (Pre-)Exascale systems like Aurora¹⁴, which will be based on Intel's X^e architecture, and Frontier¹⁵ with purpose-build AMD GPUs, a wider variability of architectures becomes available. We will see systems using NVIDIA GPUs and Intel, AMD, IBM POWER and even Arm based CPUs, Intel Xeon with X^e accelerators and completely AMD-based systems with EPYC CPUs and Radeon GPUs. Portability and maintainability of GPU applications will become more important, so developers might switch to more portable programming models like OpenMP or SYCL or even a higher-level abstraction model like Kokkos, to ensure performance portability. Tools will have to adapt to this increased variability and provide better support for more architectures and programming models.

¹²https://fz-juelich.de/ias/jsc/EN/Research/HPCTechnology/ExaScaleLabs/NVLAB/_node.html

¹³<https://www.vi-hps.org/training/tws/tuning-workshop-series.html>

¹⁴<https://press3.mcs.anl.gov/aurora/>

¹⁵<https://www.olcf.ornl.gov/frontier/>

Acknowledgments

Parts of this work have received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 824080.

The authors would like to thank the HPC performance tools community in general and the developers of the presented tools in particular as well as Andreas Herten (JSC) and Jiri Kraus (NVIDIA) from the joint JSC/NVIDIA Application Lab for many fruitful discussions, a long-standing collaboration, and the exchange of ideas.

This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.

References

1. Adhianto, L., Banerjee, S., Fagan, M., et al.: HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22(6), 685–701 (2010), DOI: 10.1002/cpe.1553
2. Beckingsale, D.A., Burmark, J., Hornung, R., et al.: RAJA: Portable Performance for Large-Scale Scientific Applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC, P3HPC, 22-22 Nov. 2019, Denver, CO, USA. pp. 71–81. IEEE (2019), DOI: 10.1109/P3HPC49587.2019.00012
3. Benedict, S., Petkov, V., Gerndt, M.: Periscope: An online-based distributed performance analysis tool. In: *Tools for High Performance Computing 2009*, Sept. 2009, Dresden, Germany. pp. 1–16. Springer (2010), DOI: 10.1007/978-3-642-11261-4_1
4. Bernat, A.R., Miller, B.P.: Anywhere, any-time binary instrumentation. In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, Szeged, Hungary. pp. 9–16. Association for Computing Machinery, New York, NY, USA (2011), DOI: 10.1145/2024569.2024572
5. Bradley, T.: GPU performance analysis and optimisation. In: NVIDIA Corporation (2012)
6. Chabbi, M., Murthy, K., Fagan, M., et al.: Effective sampling-driven performance tools for GPU-accelerated supercomputers. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 17-22 Nov. 2013, Denver, CO, USA. pp. 1–12. IEEE (2013), DOI: 10.1145/2503210.2503299
7. Cramer, T., Dietrich, R., Terboven, C., et al.: Performance analysis for target devices with the OpenMP tools interface. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, 25-29 May 2015, Hyderabad, India. pp. 215–224. IEEE (2015), DOI: 10.1109/IPDPSW.2015.27
8. Dietrich, R., Juckeland, G., Wolfe, M.: OpenACC programs examined: a performance analysis approach. In: 2015 44th International Conference on Parallel Processing, 1-4 Sept. 2015, Beijing, China. pp. 310–319. IEEE (2015), DOI: 10.1109/ICPP.2015.40

9. Dietrich, R., Tschüter, R.: A generic infrastructure for OpenCL performance analysis. In: 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS, 24-26 Sept. 2015, Warsaw, Poland. vol. 1, pp. 334–341. IEEE (2015), DOI: 10.1109/IDAACS.2015.7340754
10. Dietrich, R., Tschüter, R., Cramer, T., et al.: Evaluation of Tool Interface Standards for Performance Analysis of OpenACC and OpenMP Programs. In: Tools for High Performance Computing 2015, Sept. 2015, Dresden, Germany. pp. 67–83. Springer, Cham (2016), DOI: 10.1007/978-3-319-39589-0_6
11. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing* 74(12), 3202–3216 (2014), DOI: 10.1016/j.jpdc.2014.07.003
12. Eichenberger, A., Mellor-Crummey, J., Schulz, M., et al.: OMPT and OMPD: OpenMP tools application programming interfaces for performance analysis and debugging. In: International Workshop on OpenMP, IWOMP 2013 (2013)
13. Eichenberger, A.E., Mellor-Crummey, J., Schulz, M., et al.: OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis. In: OpenMP in the Era of Low Power Devices and Accelerators, IWOMP 2013, 16-18 Sept. 2013, Canberra, ACT, Australia. pp. 171–185. Springer, Berlin, Heidelberg (2013), DOI: 10.1007/978-3-642-40698-0_13
14. Eschweiler, D., Wagner, M., Geimer, M., et al.: Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In: PARCO. vol. 22, pp. 481–490 (2011), DOI: 10.3233/978-1-61499-041-3-481
15. Feld, C., Convent, S., Hermanns, M.A., et al.: Score-P and OMPT: Navigating the Perils of Callback-Driven Parallel Runtime Introspection. In: International Workshop on OpenMP, IWOMP 2019, 11-13 Sept. 2019, Auckland, New Zealand. pp. 21–35. Springer (2019), DOI: 10.1007/978-3-030-28596-8_2
16. Geimer, M., Wolf, F., Wylie, B.J., et al.: The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* 22(6), 702–719 (2010), DOI: 10.1002/cpe.1556
17. Gerfin, G., Venkataraman, V.: Debugging Experience with CUDA-GDB and CUDA-MEMCHECK. In: GPU Technology Conference, GTC (2012)
18. Gottbrath, C., Lüdtke, R.: Debugging CUDA Accelerated Parallel Applications with TotalView (2012)
19. Hammond, S.D., Trott, C.R., Ibanez, D., et al.: Profiling and Debugging Support for the Kokkos Programming Model. In: International Conference on High Performance Computing, 28 June 2018, Frankfurt/Main, Germany. pp. 743–754. Springer (2018), DOI: 10.1007/978-3-030-02465-9_53
20. Iyer, K., Kiel, J.: GPU Debugging and Profiling with NVIDIA Parallel Nsight. In: Game Development Tools, pp. 303–324. AK Peters/CRC Press (2016)

21. January, C., Byrd, J., Oró, X., et al.: Allinea MAP: Adding Energy and OpenMP Profiling Without Increasing Overhead. In: Tools for High Performance Computing 2014. pp. 25–35. Springer, Cham (2015), DOI: 10.1007/978-3-319-16012-2_2
22. Knüpfer, A., Brunst, H., Doleschal, J., et al.: The vampir performance analysis tool-set. In: Tools for High Performance Computing, July 2008, Stuttgart, Germany. pp. 139–155. Springer (2008), DOI: 10.1007/978-3-540-68564-7_9
23. Knüpfer, A., Rössel, C., an Mey, D., et al.: Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Proc. of the 5th Int’l Workshop on Parallel Tools for High Performance Computing, Sept. 2011, Dresden, Germany. pp. 79–91. Springer (2012)
24. Kraus, J.: CUDA Pro Tip: Generate Custom Application Profile Timelines with NVTX. <https://devblogs.nvidia.com/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/> (2013)
25. Lawrence Livermore National Laboratory: Sierra. <https://computing.llnl.gov/computers/sierra> (2020)
26. Malony, A.D., Biersdorff, S., Shende, S., et al.: Parallel performance measurement of heterogeneous parallel systems with gpus. In: Proceedings of the International Conference on Parallel Processing, ICPP 2011, 13-16 Sept. 2011, Taipei, Taiwan. pp. 176–185. IEEE (2011), DOI: 10.1109/ICPP.2011.71
27. Mayanglambam, S., Malony, A.D., Sottile, M.J.: Performance measurement of applications with GPU acceleration using CUDA. Advances in Parallel Computing 19, 341–348 (2010), DOI: 10.3233/978-1-60750-530-3-341
28. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 3.1 (2015), <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>
29. Messina, P.: The exascale computing project. Computing in Science & Engineering 19(3), 63–67 (2017), DOI: 10.1109/MCSE.2017.57
30. Mohr, B.: Scalable parallel performance measurement and analysis tools – state-of-the-art and future challenges. Supercomputing Frontiers and Innovations 1(2) (2014), DOI: 10.14529/jsfi140207
31. Mucci, P.J., Browne, S., Deane, C., et al.: PAPI: A portable interface to hardware performance counters. In: Proceedings of the department of defense HPCMP users group conference. vol. 710, pp. 7–10 (1999)
32. Nickolls, J., Buck, I., Garland, M., et al.: Scalable parallel programming with CUDA. Queue 6(2), 40–53 (2008), DOI: 10.1145/1365490.1365500
33. Oak Ridge National Laboratory: Summit. <https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/> (2020)
34. OpenACC-Standard.org: The OpenACC Application Programming Interface 2.6 (2017), <https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf>

35. OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 4.0 (2013), <https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>
36. OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 5.0 (2018), <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>
37. Pillet, V., Labarta, J., Cortes, T., et al.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: transputer and occam developments. vol. 44, pp. 17–31. CiteSeer (1995)
38. Reyes, R.: Codeplay contribution to DPC++ brings SYCL support for NVIDIA GPUs. <https://www.codeplay.com/portal/02-03-20-codeplay-contribution-to-dpcpp-brings-sycl-support-for-nvidia-gpus> (2020)
39. Saviankou, P., Knobloch, M., Visser, A., et al.: Cube v4: From Performance Report Explorer to Performance Analysis Tool. *Procedia Computer Science* 51, 1343–1352 (2015), DOI: 10.1016/j.procs.2015.05.320
40. Servat, H., Llort, G., Giménez, J., et al.: Detailed performance analysis using coarse grain sampling. In: European Conference on Parallel Processing, 25-28 Aug. 2009, Delft, The Netherlands. pp. 185–198. Springer, Berlin, Heidelberg (2009), DOI: 10.1007/978-3-642-14122-5_23
41. Shende, S., Chaimov, N., Malony, A., et al.: Multi-Level Performance Instrumentation for Kokkos Applications using TAU. In: 2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools, ProTools, 17 Nov. 2019, Denver, CO, USA. pp. 48–54. IEEE (2019), DOI: 10.1109/ProTools49597.2019.00012
42. Shende, S.S., Malony, A.D.: The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 20(2), 287–311 (2006), DOI: 10.1177/1094342006064482
43. Wienke, S., Springer, P., Terboven, C., et al.: OpenACC – first experiences with real-world applications. In: European Conference on Parallel Processing, Euro-Par 2012, 27-31 Aug. 2012, Rhodes Island, Greece. pp. 859–870. Springer (2012), DOI: 10.1007/978-3-642-32820-6_85