Trace-based Performance Analysis for Hardware Accelerators

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Diplom-Ingenieur für Informationssystemtechnik Guido Juckeland
geboren am 7. Juli 1979 in Nordhausen

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Wolfgang E. Nagel

Dresden, 8. Oktober 2012

To J.

Kurzfassung

Diese Dissertation zeigt, wie der Ablauf von Anwendungsteilen, die auf Hardwarebeschleuniger ausgelagert wurden, als Programmspur mit aufgezeichnet werden kann. Damit wird die bekannte Technik der Leistungsanalyse von Anwendungen mittels Programmspuren so erweitert, dass auch diese neue Parallelitätsebene mit erfasst wird.

Die Beschränkungen von Computersystemen bezüglich der elektrischen Leistungsaufnahme hat zu einer steigenden Anzahl von hybriden Computerarchitekturen geführt. Sowohl Hochleistungsrechner, aber auch Arbeitsplatzcomputer und mobile Endgeräte nutzen heute Hardwarebeschleuniger um rechenintensive, parallele Programmteile auszulagern und so den skalaren Hauptprozessor zu entlasten und nur für nicht parallele Programmteile zu verwenden. Dieses Ausführungsschema ist typischerweise asynchron: der Skalarprozessor kann, während der Hardwarebeschleuniger rechnet, selbst weiterarbeiten.

Die Leistungsanalyse-Werkzeuge der Hersteller von Hardwarebeschleunigern decken den Standardfall (ein Host-System mit einem Hardwarebeschleuniger) sehr gut ab, scheitern aber an einer Unterstützung von hochparallelen Rechnersystemen. Die vorliegende Dissertation untersucht, in wie weit auch multi-hybride Anwendungen die Aktivität von Hardwarebeschleunigern aufzeichnen können. Dazu wird die vorhandene Methode zur Erzeugung von Programmspuren für hochparallele Anwendungen entsprechend erweitert.

In dieser Untersuchung wird zuerst eine allgemeine Methodik entwickelt, mit der sich für jede API-gestützte Hardwarebeschleunigung eine Programmspur erstellen lässt. Darauf aufbauend wird eine eigene Programmierschnittstelle entwickelt, die es ermöglicht weitere leistungsrelevante Daten aufzuzeichnen. Die Umsetzung dieser Schnittstelle wird am Beispiel von NVIDIA CUPTI darstellt. Ein weiterer Teil der Arbeit beschäftigt sich mit der Darstellung von Programmspuren, welche Aufzeichnungen von den unterschiedlichen Parallelitätsebenen enthalten. Um die Einschränkungen klassischer Leistungsprofile oder Zeitachsendarstellungen zu überwinden, wird mit den parallelen Programmablaufgraphen (PPFGs) eine neue graphenbasisierte Darstellungsform eingeführt. Dieser neuartige Ansatz zeigt eine Programmspur als eine Folge von Programmzuständen mit gemeinsamen und unterchiedlichen Abläufen. So können divergierendes Programmverhalten und Lastimbalancen deutlich einfacher lokalisiert werden.

Die Arbeit schließt mit der detaillierten Analyse von PIConGPU – einer multi-hybriden Simulation aus der Plasmaphysik –, die in großem Maße von den in dieser Arbeit entwickelten Analysemöglichkeiten profiert hat.

Abstract

This thesis presents how performance data from hardware accelerators can be included in event logs. It extends the capabilities of trace-based performance analysis to also monitor and record data from this novel parallelization layer.

The increasing awareness to power consumption of computing devices has led to an interest in hybrid computing architectures as well. High-end computers, workstations, and mobile devices start to employ hardware accelerators to offload computationally intense and parallel tasks, while at the same time retaining a highly efficient scalar compute unit for non-parallel tasks. This execution pattern is typically asynchronous so that the scalar unit can resume other work while the hardware accelerator is busy.

Performance analysis tools provided by the hardware accelerator vendors cover the situation of one host using one device very well. Yet, they do not address the needs of the high performance computing community. This thesis investigates ways to extend existing methods for recording events from highly parallel applications to also cover scenarios in which hardware accelerators aid these applications.

After introducing a generic approach that is suitable for any API based acceleration paradigm, the thesis derives a suggestion for a generic performance API for hardware accelerators and its implementation with NVIDIA CUPTI. In a next step the visualization of event logs containing data from execution streams on different levels of parallelism is discussed. In order to overcome the limitations of classic performance profiles and timeline displays, a graph-based visualization using Parallel Performance Flow Graphs (PPFGs) is introduced. This novel technical approach is using program states in order to display similarities and differences between the potentially very large number of event streams and, thus, enables a fast way to spot load imbalances.

The thesis concludes with the in-depth analysis of a case-study of PIConGPU—a highly parallel, multi-hybrid plasma physics simulation—that benefited greatly from the developed performance analysis methods.

Contents

Nomenclature

API	An application programming interface (API) specifies how software components can communicate with one another. It typically defines routines and data structures.
ASIC	An application specific integrated circuit (ASIC) is a special purpose processor designed to perform one specific task very well. It cannot execute any arbitrary computable algorithm.
DDR	Double data rate (DDR) memory is a special form of computer memory that can transmit data on both the rising and falling edge of the clock signal.
die	The die is one coherent piece of semiconductor substrate that contains electric circuitary.
DirectCompute	DirectCompute is a component of Microsoft's DirectX which allows non-graphical computing of data that is stored in the graphics accelerator's memory.
DMA	Direct memory access (DMA) is a technique that allows the mapping of two disjoint address spaces into one region so that one process can directly access data from another. It is typically used for data transfers from and to external devices.
GUI	A graphical user interface (GUI) enables an interaction between a computer program and its user using images and gestures (via a mouse or direct using a finger) rather then text commands.
IDE	An integrated development environment (IDE) is a GUI based piece of software that contains or integrates all tools necessary for the development of software. It usually contains a source code editor, an automated software build tool, and a debugger. Some IDEs also contain performance analysis tools.
Inlining	Inlining is a technique used by compilers which copies small subroutines directly into the place where they were called to save the overhead for a subroutine invocation.
ISA	An instruction set architecture (ISA) defines an interface between soft- and hardware. It specifies supported data types and operations, provides a format for the operations, and indicates how memory can be accessed.
MIPS	MIPS is an acronym for "microprocessor without interlocked pipeline stages" and describes a special reduced instruction set computer (RISC) ISA.
MPI	The message passing interface (MPI) is a very popular API for inter-node communication for parallel applications.

OpenMP	Open Multiprocessing is a directive based multi-threading API whose implementation enables shared memory communication between threads spawned by the application.
PCI-E	see *PCI-Express*
PCI-Express	PIC-Express is a technology to connect external devices such as graphics or network adapters to the central processing unit (CPU).
PRACE	The partnership for advanced computing in Europe (PRACE) is an organization that unites high-end supercomputing centers with the goal to provide a highly capable computing infrastructure for running very complicated simulations.
SDK	A software development kit (SDK) is a set of software tools that aid the programming of applications for a specific software package.
SIMD	A popular concept to produce more than one result with one instruction is single-instruction, multiple-data (SIMD) computing. One operation is carried out on multiple data words.
VLIW	A very long instruction word (VLIW) ISA bundles multiple instructions into one instruction word. One instruction word is the smallest unit that the processor can digest. VLIW architectures usually have a fixed mapping of instructions in the instruction word to the function units in the hardware.

1 Introduction

Hardware accelerators have changed how people perceive computing. Mobile devices aided by cloud services have revolutionized the common usage of digital media and computing services. Another trend even before the accelerators were multi-core processors. Both require parallel computing to utilize the full performance of the chip. As a result, parallel programming moved from the supercomputing niche to the main stream. The aforementioned trends also have resulted in an increasing number of highly parallel, low power single chip platforms. Even high performance computing (HPC) is starting to embrace hardware accelerators as a way for a better ratio between performance and power consumption.

Performance analysis and optimization has always been part of developing leading edge parallel applications. Especially high performance computing applications with their inherent parallelism offer a high potential for increased performance after a better understanding of the program's behavior. Sample based profiling and event logging are both well established techniques in determining this behavior, albeit with a different detail resolution. If profiling is considered a magnifying glass which provides a closer view at the application, event logging will be the microscope allowing a very detailed look into the inner workings of the program.

Performance analysis tools cover the traditional layers of parallelism (inter- and intra-node parallelism) very well. Yet at the same time, many HPC tools do not cover the new kind of parallelism introduced by the hardware accelerators. This is unfortunate, since the interaction of all layers is the key to high performing, scalable applications on these novel HPC architectures. The focal point of this thesis is the question, how performance analysis tools can include this new dimension in parallelism and to provide all necessary data to a performance analyst so that the application performance can be optimized. The diversity of the available hardware accelerators complicates an easy answer to that question and requires the establishment of a common ground between them first. On this foundation, a performance tool interface for the various accelerators can be built. As a result both hardware accelerator specific performance studies but also the analysis of whole systems becomes possible.

Contributions of this thesis

This thesis delivers the following contributions to the field of performance analysis of parallel applications:

- A platform independent performance tool interface that provides access to performance relevant data directly from the device, such as time stamps and hardware performance counter values is developed. One implementation of this proposed interface was developed together with NVIDIA and is now available as the CUDA Profiling Tool Interface (CUPTI).
- A novel visualization approach for event logs is proposed. Parallel program flow graphs remove

the time stamps from the event data and aggregate them into program states and their durations. The graph can display common and different program states to easily show the actual structure of the application.

- A generic method for recording event data from any kind of API driven hardware accelerator using an automated library interception is presented. This technique can be applied to unmodified applications and can also be used for non-accelerator APIs.

Conceptual Formulation and Structure of the thesis

This thesis investigates the possibilities for application performance analysis of hardware accelerated programs. It develops multiple models and techniques for recording and displaying various aspects of the execution of such programs in order to help the programmer understand how the application is executed.

As a first step this thesis introduces the hard- and software ecosystem of a computer that features hardware accelerators in Chapter 2. In the following Chapter 3 the current state of performance data monitoring, recording, and visualization is presented. This chapter also contains a brief introduction of other performance tools that also target hardware accelerators. Using this foundation the principles of performance data acquisition using event logs is extended to include hardware accelerator APIs in Chapter 4. Furthermore, it is discussed how hardware accelerator performance data can be visualized as an additional layer of concurrency in hybrid parallel programs. The findings from all previous chapters are combined in Chapter 5 where the methods, models, and techniques will be applied to one selected highly complex GPU accelerated scientific application. The thesis is concluded with a summary of the presented work and an outlook into further research.

The following text formatting is used throughout the thesis to highlight various semantic aspects and to ease reading of the text: *Italics* for terms that are explained in more detail in the Nomenclature, SMALL CAPS for authors, and `non-proportional font` for function names and program segments.

2 Hardware Accelerators

The history of hardware accelerators goes back almost as far as the history of the von-Neumann computer architecture itself. While a general purpose processor—also called central processing unit (CPU)—offers the flexibility to execute any computable algorithm, it in return lacks the speed of a specialized processing unit dedicated for one specific purpose. One might argue that the very early incarnations such as specialized input/output or memory controllers, which indeed accelerated overall computation, actually qualify to be termed "hardware accelerators". If one considers including coprocessors and their general definition:

> **Coprocessor**: Additional processor used in some personal computers to perform specialized tasks such as extensive arithmetic calculations or processing of graphical displays. The coprocessor is often designed to do such tasks more efficiently than the main processor, resulting in far greater speeds for the computer as a whole. [Bri11]

it will become clear that almost every device can be a hardware accelerator. The main difference between a coprocessor and a hardware accelerator is the capability to execute own programs. While a coprocessor, e.g. a floating point unit, receives its instructions from the CPU, a hardware accelerator actually can run its own program. Its execution, however, is controlled by the CPU, such that the CPU determines when the "external" program is launched, provides input data, and gathers output. As a result one can reason that coprocessors are one special form of hardware accelerators.

A very popular example of a hardware accelerator as of 2012 is a graphics processing unit (GPU). Originally designed as video display coprocessors e.g. in the Atari 8-bit family, they quickly evolved into semi-independent processing units converting raw data supplied by the CPU (polygon meshes and textures) into a photo-realistic three-dimensional depiction. Other currently available hardware accelerators include compute accelerators or interface accelerators (such as special DSP enhanced sound cards or network interface cards). This thesis will focus on the former since it is currently for computational science of great interest to utilize the potential of a heterogeneous, hardware accelerated computer architecture.

This chapter first introduces a few special purpose accelerators that could not gain a wide acceptance, then introduces currently widely used accelerators. It concludes with the extraction of general principles in the interaction with the accelerators and, thus, lays a foundation for the following discussion of the possibilities for performance analysis. Each section focuses on one accelerator technology by outlining its programming model and by discussing its microarchitecture.

2.1 Rise and Fall of Special Purpose Solutions

The history of hardware accelerators has seen technologies that seemed promising, but ultimately not gained enough market share or were incorporated into either the CPU or other accelerators. A popular example is the company "3dfx Interactive" whose Voodoo graphics accelerators enabled a leap in photorealism in computer games and computer animation in general, but eventually were also available in "standard" graphics cards [NVI00]. The same holds true for Ageia PhysX which was integrated into NVIDIA [NVI08].

Two current hardware accelerator platforms that are discontinued or have not made a significant impact are studied exemplary in this section: the Cell Broadband Engine and Clearspeed Accelerators.

2.1.1 Cell Broadband Engine

The IBM Cell Broadband Engine (Cell/B.E.) is a heterogeneous multi-core processor. Its programming model and the processor microarchitecture, however, share a lot with other accelerators, hence it is included in this section. The Cell Broadband Engine Architecture (CBEA) as the underlying *ISA* was designed as a novel processor architecture with a strong focus on performance and energy efficiency while at the same time requiring a very low level programming approach [KDH+05]. In order to achieve this the CBEA does not use out-of-order execution, speculation, deep pipelines, or large caches. The architecture, a joint development by Sony, IBM, and Toshiba, was developed for the Sony Playstation 3 game console, which could not connect to the success of its predecessor, and special Toshiba TV appliances, which never made it to market. Its—at the time—outstanding floating point computing capabilities quickly attracted the interest of computational simulation scientists who deemed the subsidized Playstations as a very cheap way to crunch numbers. As a consequence, IBM also offered the Cell/B.E. as a component in its own product portfolio which overcame some of the hardware limitations of the Sony Playstation 3. While this first version had a large discrepancy between single and double precision floating point performance, which was of no real importance for computer gaming or multimedia entertainment, a minor revision of the processor—the IBM PowerXCell 8i—compensated this "flaw" [BDH+08]. The processors fueled the fastest supercomputer of the world from June 2008 until November 2009.

The Cell/B.E. combines a lightweight standard PowerPC CPU (Power Processor Unit, PPU) with up to eight special purpose processing elements (Synergetic Processing Units, SPUs) as shown in Figure 2.1. While the SPUs carry the computational workload, the PPU is responsible for launching and monitoring the work of the SPEs.

Cell/B.E. Programming Model

The programming model for the Cell/B.E. stipulates separate binary streams for the PPU and SPUs. While both offer the capability to work with standard C programs and special libraries for on-chip communication, especially the handling of the Local Store and code scheduling on the SPE required a lot of fine-tuning to achieve a close-to-maximum performance [EOO+06]. The SPU code is embedded into the PPU binary so that the PPU can execute and launch the work on the SPUs (see Figure 2.2) As a result

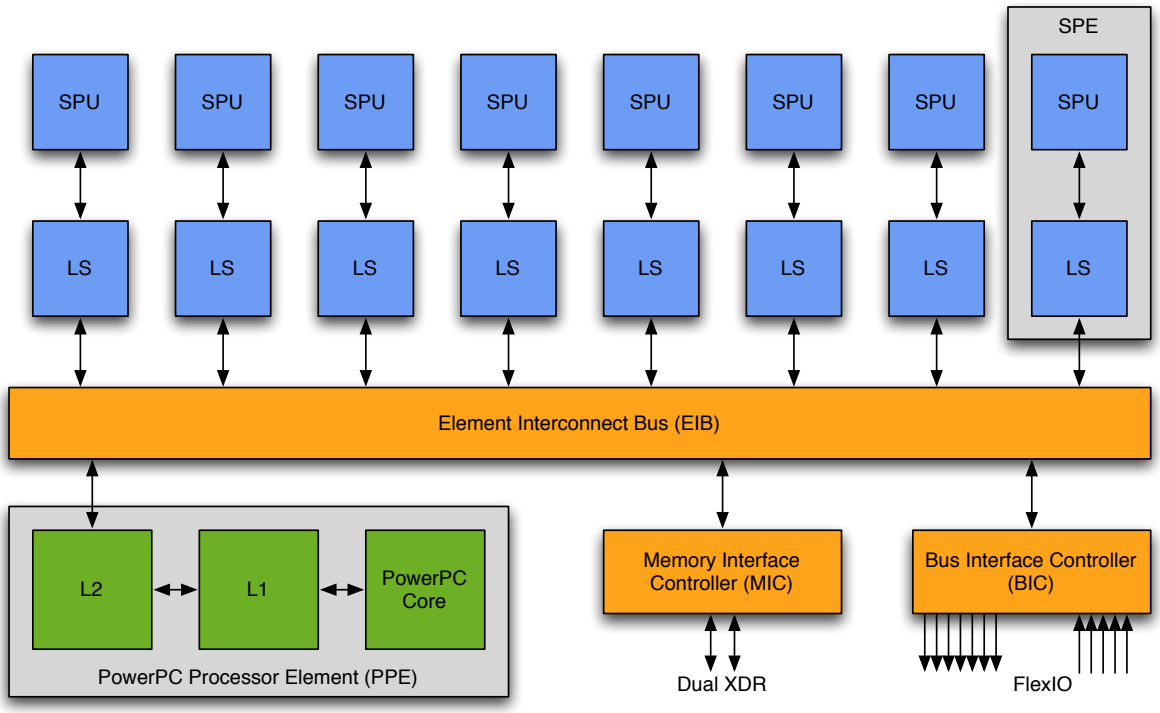

Figure 2.1: The block diagram of the Cell Broadband Engine shows how all major components of the processor are grouped around the Element Interconnect Bus. The EIB can, if correctly used, enable all chip components to communicate concurrently without inducing a bottleneck. [Hac07]

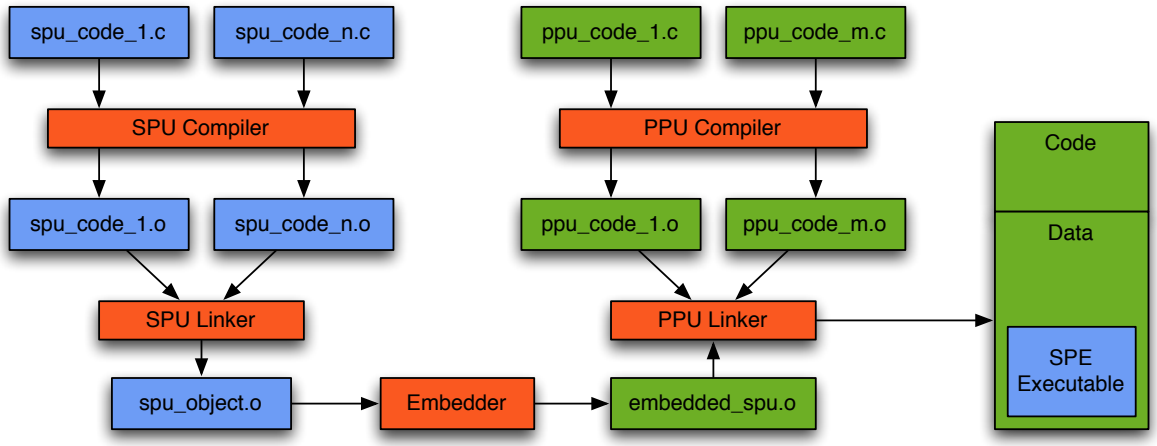

Figure 2.2: The compilation and linking of a Cell/B.E. executable requires multiple steps since the the application is started on the PPE. The PPE binary contains the SPU executables for all parts that will be offloaded onto the SPEs as binary data objects. The whole process is aided by the compiler tools provided by the vendor. [Hac07]

of this rather complex programming model the number of scientific applications that outperform their counterparts on "classical" HPC environments is rather limited [LANL] and the Cell/B.E. could not gain a strong foothold in simulation sciences as a high performance computing platform.

Cell/B.E. Architecture

As already shown in Figure 2.1 the key elements of the CBEA are:

- The Synergetic Processing Units (SPUs) which, combined with a small 256 kB self-addressable cache called Local Store (LS), form the eight Synergetic Processing Elements (SPEs) of the Cell processor. The SPUs also offer SIMD parallelism by allowing operations on packed 128 bit vectors. With a theoretical peak performance of—depending on the clock speed—up to 25.6 GFLOPS (at 3.2 GHz) per SPE in single or 12.6 GFLOPS in double precision (for the revised PowerXCell 8i) they provide the horse power and precision to drive computation at a very high speed.
- The Power Processing Element (PPE) is the controller for the SPEs which starts, monitors, and stops work on them. Due to its similarity to the standard IBM POWER processors it is also capable to run a standard operating system such as a standard GNU/Linux.
- The Element Interconnect Bus (EIB) is a two-lane bi-directional network ring that connects not only the SPUs and the PPE but also the memory controller and the I/O interfaces. Due to its capability to utilize only sections of the ring and then in turn have multiple transfers on one physical ring, it reaches a total transaction bandwidth of 204.8 GB/s. For typical data streaming applications this is a sufficient bandwidth to saturate the data requirements of all SPEs.
- The first version of the Cell/B.E. uses a Rambus memory subsystem with a dual channel Rambus XIO interface and Rambus XDR memory. This not widely used technology offers a very high memory bandwidth of up to 25.6 GB/s. At the same time it is limited in the total amount of memory and very expensive. The PowerXCell 8i uses standard DDR2 memory to overcome these shortcomings.

Performance studies in [Hac07] demonstrated the capabilities of the Cell/B.E.. Specifically designed microbenchmarks were able to achieve the maximum compute and memory transfer capabilities of the processor (see Figures 2.3 and 2.4). At the same time it is also argued that the migration of real applications requires a large amount of work and, thus, rather small plugins, e.g. H264 video decoding, edge detection, and face recognition, are implemented and put together to form more complex workflows.

In retrospect, the Cell/B.E. has a number of features that are common to hardware accelerators in scientific computing—as will be shown in the remainder of this chapter: A limited amount of available memory, a self-controlled very fast cache per processing element, and very high data transfer rates to and from all components of the chip.

2.1.2 ClearSpeed Accelerators

The ClearSpeed CSX accelerators are made of massive SIMD processors. Its current incarnation, the CSX700 processor, offers up to 96 GFLOPS of double precision IEE754 compliant floating point per-

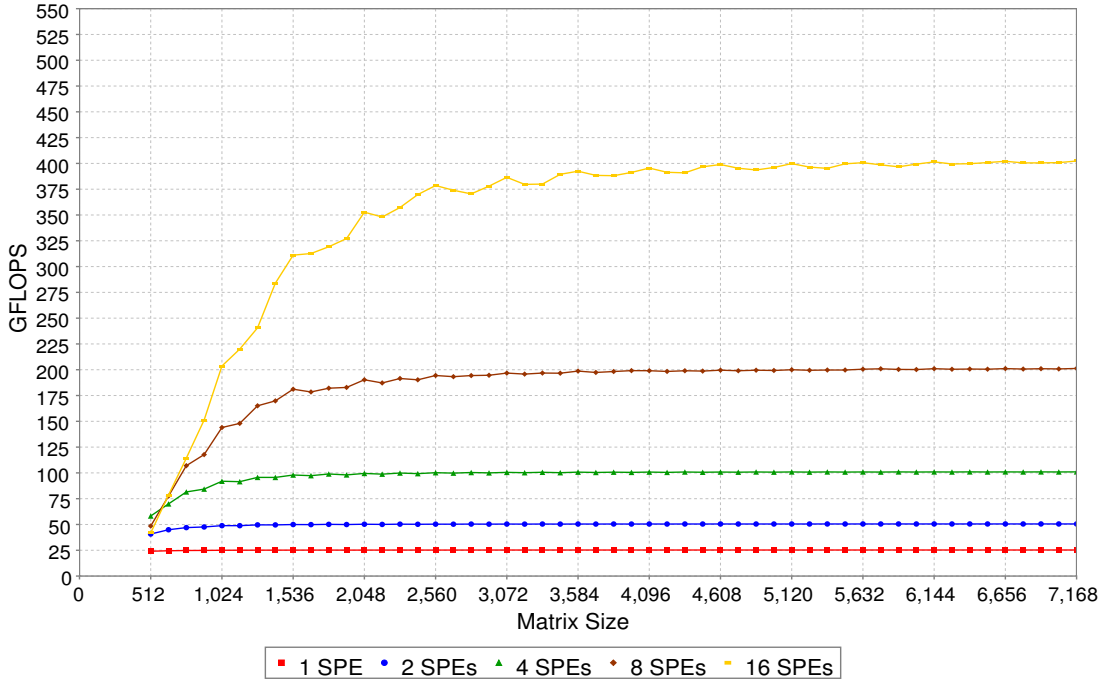

Figure 2.3: The performance measurement for the matrix-matrix-multiplication on the Cell/B.E. demonstrates that the processor can deliver its peak performance for each SPE. It does, however, take very large matrix sizes to saturate all compute elements with enough work. [Hac07]

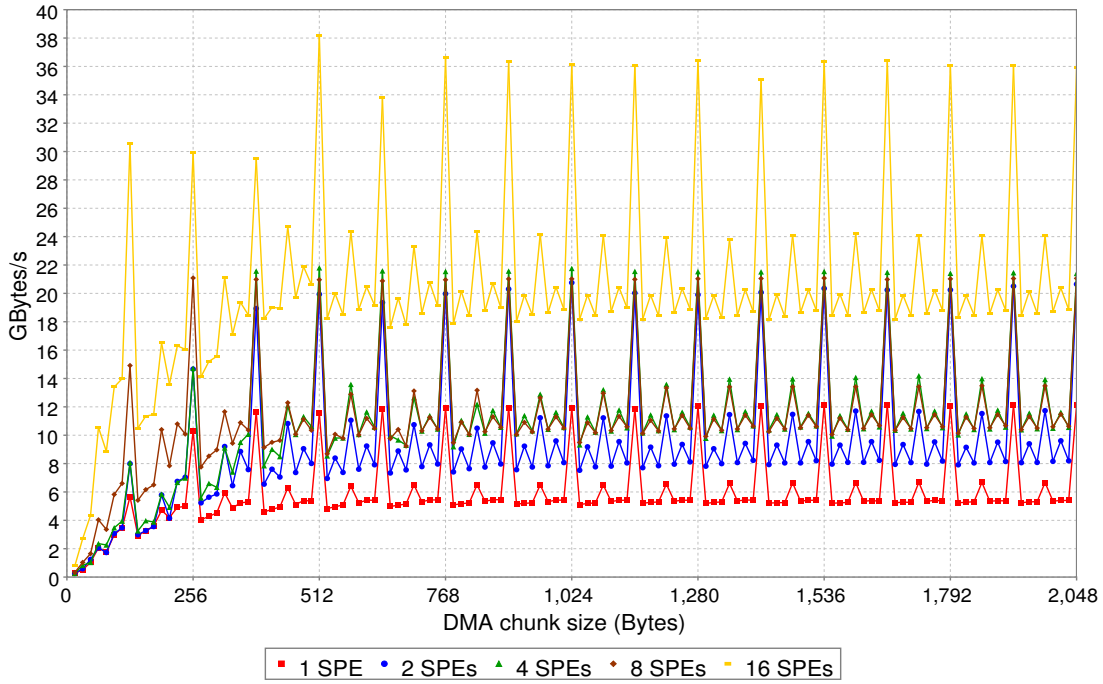

Figure 2.4: The performance measurement of DMA memory transfers on Cell/B.E. shows the same scalability as the compute capabilites of the device. It is to note, however, that the achieved performance depends highly on the transmitted data chunk size. [Hac07]

formance at only 9 Watt power consumption [Cle11]. It is typically available as a *PCI-Express* extension card which contains the CSX700 processor and 2 GByte RAM. The ClearSpeed architecture was designed for high reliability and low power high performance computing from the beginning. Hence, ECC error protection for all levels of memory was included from the very first product onwards.

ClearSpeed Programming Model

The ClearSpeed architecture differentiates between scalar (mono) and parallel (poly) workloads [Cle08]. The scalar work is executed on the mono execution units which also decode any parallel commands before sending them to the poly execution units. In order to achieve maximum performance, the programmer has to utilize the poly units which is done by a language extension to the C programming language called C^n. Data is placed on the mono or poly units by adding the keywords `mono` or `poly` to the declaration of variables. The 96 poly-elements of the CSX700 processor then discern themselves from one another by calling the intrinsic routine `get_penum()`. Typical library functions are provided with an apended "`p`" to their names to be executed on the poly units. The following simple example illustrates the principle:

```
#include <stdiop.h>   // Output support
#include <lib_ext.h>  // Extra functions to support features of hardware

int main() {
poly int n;
n = get_penum(); // individual PE number
printfp("PE number: %d\n", n); // Output different message per PE
return 0;
}
```

The code will result in every poly processing elements (PEs) acquiring its local ID and printing it to stdout. In the same manner work can be divided among PEs with the limitation that all PEs typically execute the same instruction (unless they are masked using `if` statements on the PE ID). This feature is very similar to the thread groups used by graphics processors as discussed in Section 2.2.3.

ClearSpeed Architecture

The CSX700 processor can execute up to two independent program streams with its two SIMD processors (see Figure 2.5). Each processor can operate on 96 elements in parallel using the 96 PEs of one SIMD processor. The processor is clocked at 250 MHz which results in two floating point operations per clock per processing element. Furthermore the processor supports multi-threading.

The CSX600 chips powered parts the Tsubame cluster of the Tokio Institute of Technology from 2006 until 2009 and, thus, were part of the then 7th fastest supercomputer in the world. Furthermore, the CSX700 chips are now available as part of special HP blades and were included in two PRACE machines in the Netherlands and France. The computing power of one ClearSpeed processor did not offer much more performance than the host CPUs, hence, could not convince programmers to use them for a performance gain.

Own measurements (see Figure 2.6) with the a CSX620—a predecessor of the CSX700 chip—show the capability of the chip to be fully utilized while at the same time offering a very good performance per

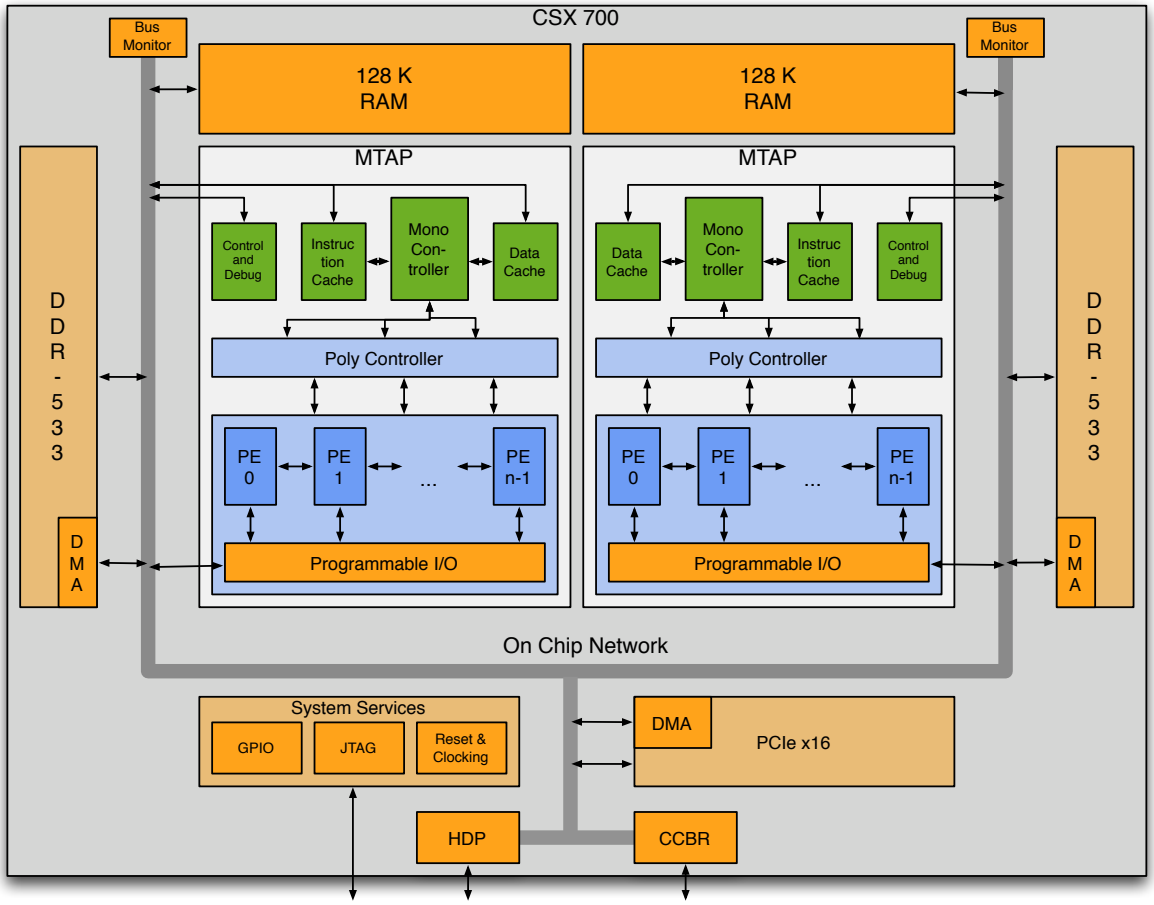

Figure 2.5: The Clearspeed CSX700 processor block diagram shows the main features of the chip: the dual processor design, each with a mono processor for serial workloads and the poly processing elements operating as a processor array[Cle11].

Watt ratio. The ClearSpeed accelerator boards also have a very limited amount of board local memory (2 GByte), emphasizing a common property of hardware accelerators.

2.2 State-of-the-Art Accelerator Technology

A very limited number of hardware accelerators enjoy a great popularity across a range of platforms. Floating point computing is nowadays common in all computing devices from smartphones to supercomputers. Graphic processing units (GPUs) have also become quite popular and have moved onto standard processors as well. Hence, successful hardware accelerators can be seen as an "early adoption" of upcoming standard processor features. A third very popular yet not directly obvious acceleration platform are reprogrammable processors (e.g. FPGAs). They are the processor of choice in an increasing number of embedded systems, since a simple software update can also fix hardware flaws. They are also of interest to a—albeit limited—group of simulation scientists who have applications that are just not suited for standard "fixed" function processors (like CPUs or GPUs). This section will provide an overview over these three technologies.

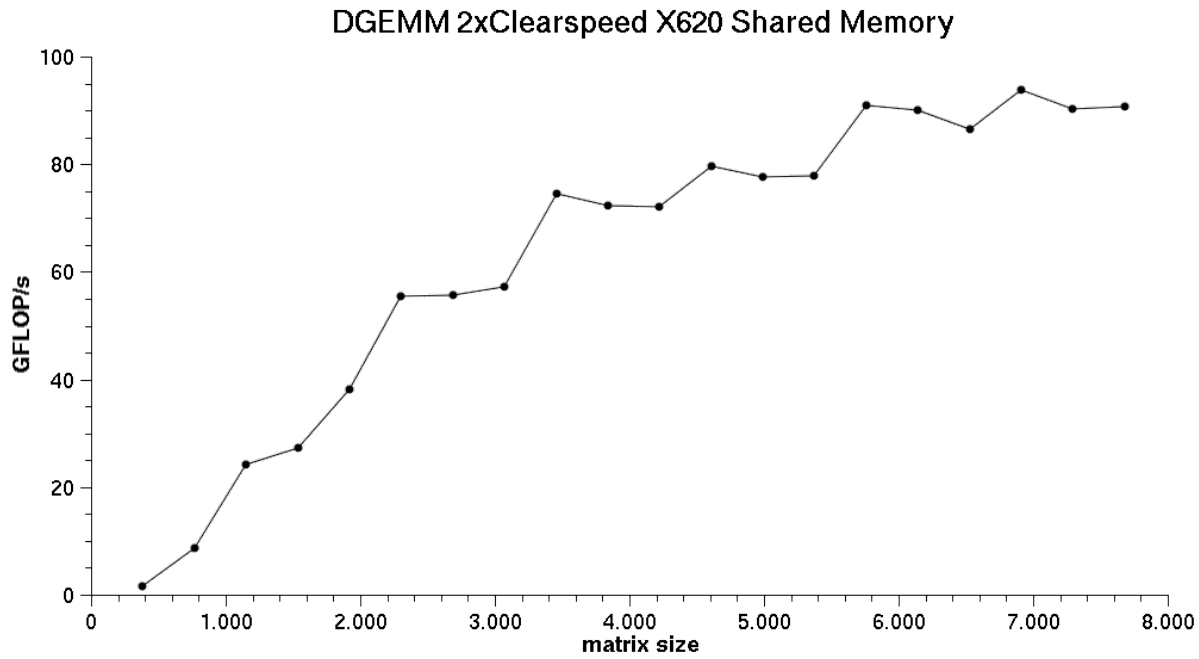

Figure 2.6: The performance of a matrix-matrix-multiplication on a ClearSpeed X620 board using the vendor provided DGEMM library implementation shows how a very large matrix size is required to fully utilze the poly compute units on the device.

2.2.1 Floating-Point Acceleration

The currently dominating desktop and HPC processor architecture is the x86 architecture. Its backwards-compatibility can be considered a blessing as well as a curse. The processor architecture starts with the Intel 8086 processor which was launched in 1978. This CPU did not have any built in floating point processing unit and, thus, had to emulate floating point arithmetic with special compiler induced emulation on fixed point ALUs [ia09]. Since the word size at that point was only 16 bits, this also leads to very inaccurate floating point computation. In order to offer a higher precision and higher throughput floating point engine, Intel announced the 8087 floating point coprocessor in 1980. Other processors, such as IBM POWER or Sun SPARC, featured floating point computation from the very beginning.

The basic features of the x87 floating point co-processor have remained the same throughout the various generations of x86 chips who always had its x87 counterpart. The growing demand for floating point computation in standard desktop applications combined with a larger number of transistor available on a CPU led to the inclusion of the coprocessor on the same *die* as the CPU with the Intel 80486 generation. After that a number of instruction set extensions to the IA-32 *ISA* [Int11] to speed up multimedia and computer gaming related operations were introduced. The extensions all share the SIMD approach, meaning that one instruction computes on up to four double precision floating point operands.

FPU Programming Model

Floating point units (FPUs) are accessed via floating point instructions that are either a part of the native *ISA* or an *ISA* extension. They typically have an own set of registers that can be of different width than

Figure 2.7: The x87 FPU execution enviroment provides an extended internal floating point format using 80 bits and eight registers [Int11].

the general purpose registers (GPRs). Due to this difference in data size, floating point units have their own load and store operations. A small example of *MIPS* assembler illustrates the usage of a floating point unit from within the code.

```
loop:
   L.D    F1,30(R2) # load double from address in R2 + 30 and store in F1
   L.D    F2,38(R2) # load double from address in R2 + 38 and store in F2
   ADD.D  F3,F1,F2  # add F1 and F2 -> store in F3
   S.D    F3,46(R2) # store double drom F3 in address in R2 + 46
   DADDIU R2,R2,#24 # add 24 (3*8) to address in R2
   DADDIU R1,R1,#-1 # decrement loop counter
   BNEZ   R1,loop   # return to top of loop if counter != 0
```

FPU Architecture

One common feature of the x87 coprocessors is the use of internal 80-bit wide registers (see Figure 2.7). The registers are organized in an 8-entry register stack and the contents are always separated as exponents and fraction which each have their own computing units. With a peak performance of about 50,000 FLOPS the 8087 coprocessor offered speedups between 20 and 500 % (depending on the used algorithm). Its current versions which are integrated as floating point units (FPUs) on the main processor itself only offer x87 as a processing mode for compatibility reasons while the *ISA* extensions have actually replaced the need for x87 instructions. Streaming SIMD extensions (SSE) are available in four generations themselves and are just followed by the AVX extension in Intel's current Sandybridge CPUs.

The main difference to the x87 mode is the capability to directly address up to 16 registers which offers much more flexibility than working with a register stack. With the integration of the FPU into the CPU, they are now also clocked at the same frequency as the main x86 processor. Due to the SIMD capabilities of processing up to four double precision floating point operands simultaneously, the performance of one FPU increased to over 10 GFLOPS.

The major difference of on-chip floating point acceleration compared to other hardware acceleration techniques is the fact, that the instructions for the floating point unit are embedded in the "normal" instruction stream of the CPU. Hence, the FPU qualifies more as a co-processor than an independent hardware accelerator. Other processor architectures like SPARC or POWER have the floating point unit as an integral part of their processor and *ISA* design from the very beginning [HP07]. The POWER and PowerPC architectures also offer a vector extension for their floating point units called AltiVec which is comparable to Intel's SSE. Internally, FPUs usually use multiple pipelines for adding, multiplying, and memory transfers of floating point data. Some FPUs also feature a fused multiply-add-unit that can run both operations in one pipeline.

The most prominent fact about FPUs is that their wide adoption ultimately led to their inclusion in the main CPU which serves as an example how far a successful hardware accelerator design can actually come.

2.2.2 Field Programmable Gate Arrays (FPGAs)

FPGAs are unique accelerators in the sense that they implement a software program in hardware. Their specific purpose is not determined when they are manufactured, but rather when they are programmed. They enjoy great popularity for *ASIC* prototyping, but are due to their decreasing prices also more and more popular in appliances instead of *ASIC*s [TSV07]. FPGAs can exist on their own and run an operating system or exist as a programmable fixed purpose unit (e.g. a hardware RAID controller for a set of disks [Net]).

For simulation scientists FPGAs are of interest whenever an algorithm only inadequately fits onto a standard processor. The flexibility of the hardware makes it possible to have the processor actually implement the algorithm, e.g. build a systolic array through which the data is streamed. Other possibilities include the design of a problem specific pipeline that includes as many steps as needed while still producing one result per clock tick [HD08].

FPGA Programming Model

The basic building blocks of FPGA hardware are look-up-tables (LUTs) which are programmable to implement any given logic function which can be expressed using boolean logic. When programming an FPGA each look-up-table is assigned the logic function to implement, a wiring plan to place and connect the LUTs is produced, and a clock specification to drive the processing on the programmed chip is provided. All this is a very low level chip design and does not differ from the process *ASIC* or also CPU designers step through in order to produce a new chip[1]. Therefore, the same tools are

[1] It does, however, not include the layout optimization prior to producing the lithography masks and the actual manufacturing.

used for programming FPGAs: hardware description languages like VHDL or Verilog. At this point the major weakness of FPGAs as hardware accelerators for scientific computing is exposed: they are almost impossible to program for a "normal" scientists working in a specific problem domain.

A very simple VHDL code describing the behavior of a logical AND-gate is shown here [VG02]:

```
library ieee;
use ieee.std_logic_1164.all;

-------------------------------------------------

entity AND_ent is
port(        x: in std_logic;
        y: in std_logic;
        F: out std_logic
);
end AND_ent;

-------------------------------------------------

architecture behav1 of AND_ent is
begin

    process(x, y)
    begin
        -- compare to truth table
        if ((x='1') and (y='1')) then
            F <= '1';
        else
            F <= '0';
        end if;
    end process;

end behav1;

architecture behav2 of AND_ent is
begin

    F <= x and y;

end behav2;
```

While the syntax is similar to C the whole system design process—designing a finite-state machine, the control flow graphs, and the data paths—is the most time consuming part of the engineering process.

Due to this high demand on the FPGA programmer, a number of high level programming approaches have been proposed in order to enable every programmer to also utilize FPGAs. The commonality between languages like Handel-C, Impulse-C, or Mitrion-C is that they all use the programming language C as a basis and extend it with own syntax and semantics. The languages introduce parallel regions where the statements within must have no data dependencies. This allows for the FPGA to unfold its potential for massive concurrency, especially when the regions are rather large. A sample Mitrion-C code that finds the maximum in a list looks like:

```
Mitrion-C 1.0;
int:32 main(int:32<100> a)
{
    int:32 bestMax = 0;
    int:32 max =  for(element in a)
    {
        bestMax = if(element > bestMax)
            element
        else
            bestMax;
    } bestMax;
} max;
```

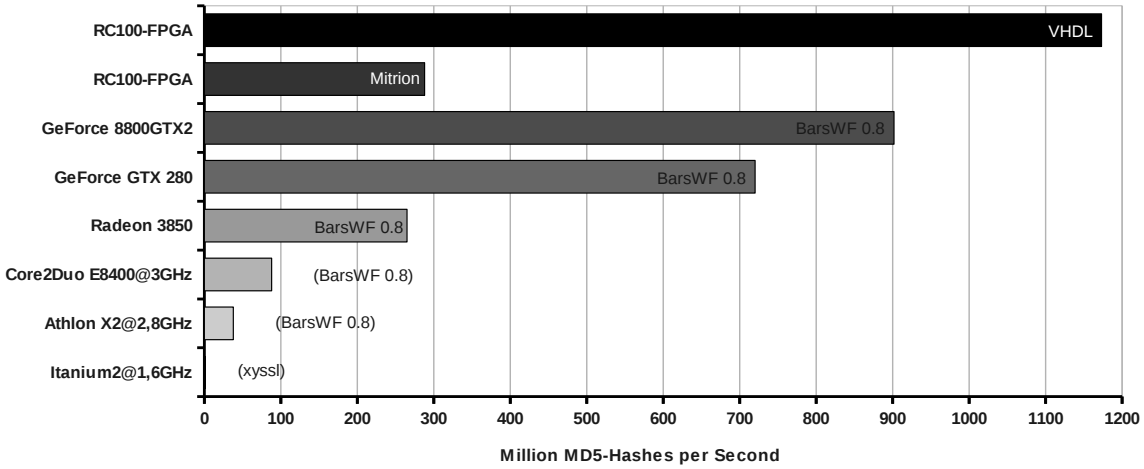

Figure 2.8: The performance of an MD5 brute-force collision detection algorithm varies quite a lot depending on the chosen hardware platform and the used implementation scheme. The fastest and most power efficient implementation used VHDL on the SGI RC100 blade. [Die09]

The relative ease of programming, however, comes at the price of performance. The high level languages have to utilize some mechanism—in case of the more closely studied Mitrion-C the virtual Mitrion processor, a flexible parallel and pipelined processor design—to actually synthesize an application specific processor model. This abstraction has drawbacks compared to a real application specific processor design using a Verilog/VHDL approach. One such drawback is the fixed clock rate of 100 MHz for the Mitrion based processor design while a HDL based design can be synthesized with any clock frequency supported by the FPGA hardware within the limits of the synthesized design.

Performance comparisons of an MD5 checksum algorithm and a solution for the N-Queens problem which both map very well to the FPGA show dramatic differences between VHDL and Mitrion-C (see Figure 2.8). The most notable features of FPGAs especially in the area of scientific computing are, again, a rather small amount of local memory attached to the processing unit and a rather limited number of applications where FPGAs can outperform the CPU or other accelerators.

A new approach to bring FPGAs into HPC was started in 2008 when the company Convey introduced its hybrid computing platform called HC-1 [Con10, Con09a]. The HC-1 places an FPGA into a CPU socket and allows for very fast FPGA-CPU-interaction and enables very fast access to the rather large main memory. Beyond that the company also delivers a software ecosystem that allows to easily implement own solutions. Own codes can be compiled into so called personalities that can then be loaded onto the FPGA by a host program. A number of pre-defined personalities allow for a turn-key solution in these areas, especially life sciences.

FPGA Architecture

Since the complexity of FPGA usage is purely in programming them, the hardware is rather simple. The typical hardware design of an FPGA consists of a large number of programmable lookup tables (LUTs), a lot of wires connecting them which in turn can be programmed to connect at their intersections, and in most cases a small number of floating point units (see Figure 2.9). The configuration of the FPGA is

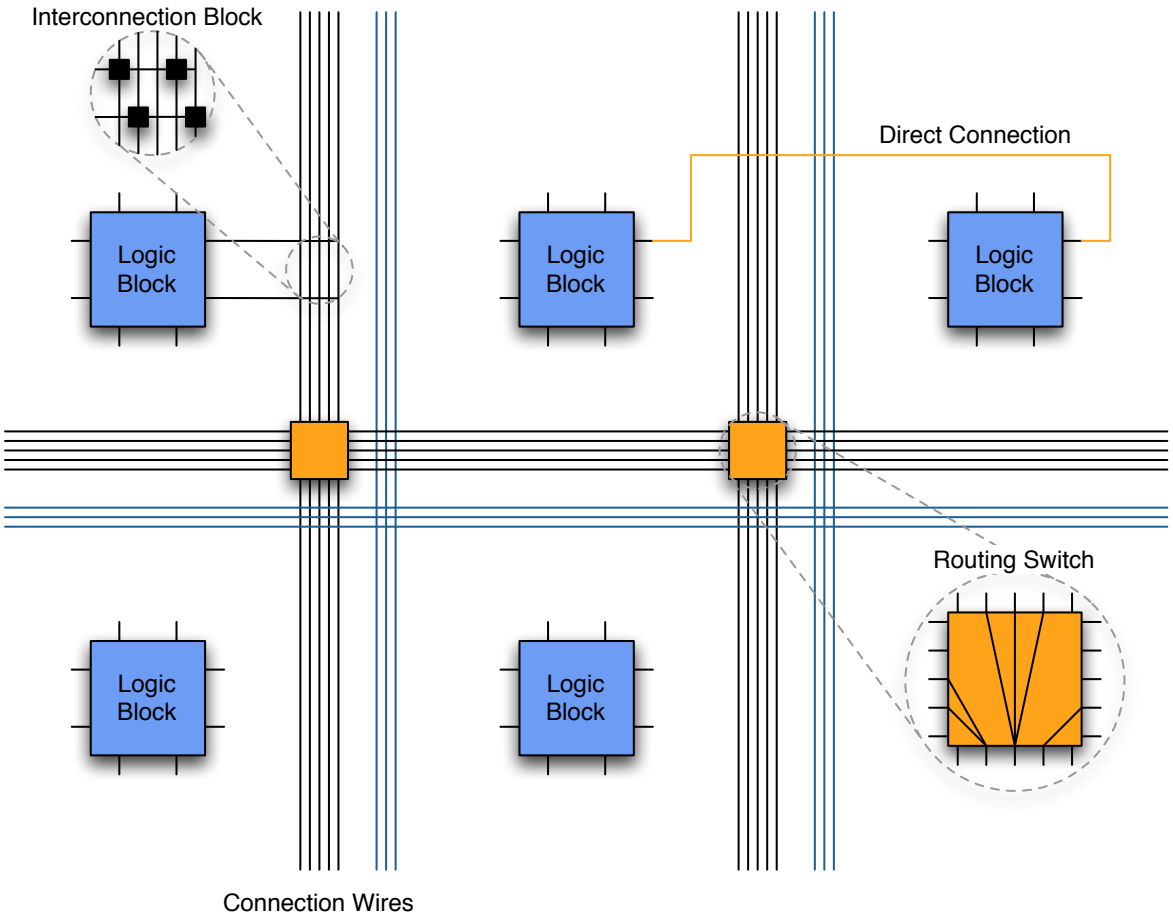

Figure 2.9: The FPGA chip archtiecture typically consists of programmable logic blocks that can be connected to one another using direct connections or via a switchable interconnection network [Die09].

created by translating the high level hardware description language before using hardware synthesis tools to map this description onto FPGA function units and connecting them. Despite the existence of floating point units on FPGAs, they cannot compare in floating point performance with standard processors or other hardware accelerators [HD08]. The most prominent FPGA manufacturers nowadays are Xilinx and Altera.

2.2.3 Graphic Processing Units

Graphic Processing Units (GPUs) are a very recent addition to the hardware accelerator landscape for scientific computing [KH10]. Dedicated graphics processors emerged at first for the Atari gaming console in the 1970s. The success of the computer entertainment industry also drove the advances in computer graphics with an increasing demand for more realism in computer games. Major milestones towards establishing graphics processors as hardware accelerators for scientific simulation as well as main stream applications were: The Commodore Amiga in the 1980s, the addition of accelerated graphic adapters to PCs in the 1990s, and the introduction of OpenGL and DirectX as graphic APIs. The turning point for GPUs as hardware accelerators for non-graphic output from a hardware perspective is the introduction

of DirectX 8 which required universal shader/texture processors that were able to run custom programs. With this foundation, the next requirement for GPU computing is a programming model that does not require in-depth knowledge of graphic programming. This was the case in 2007 when NVIDIA released its Compute Unified Device Architecture (CUDA) which extends the C programming language with new constructs to allow subroutines to execute on the GPU. One year later OpenCL—a platform independent framework for using heterogeneous computing devices (i.e. hardware accelerators)—was released. Although OpenCL has been designed not specifically for GPUs, it has a strong similarity to CUDA. Due to its usability for a multitude of accelerator devices, OpenCL will be discussed in an own section. Like many other accelerator technologies, GPUs also have their own local memory which they use to store the data currently being worked upon.

GPU Programming Model

Using graphics processors for computation that is not directly related to graphics output either via CUDA or OpenCL generally follows the same concept: the code portion which is to be accelerated is offloaded into one or more functions—called kernels—to be executed on the GPU [KH10]. Their input data has to be transferred manually into the GPU's local memory before the kernels are invoked from the host. Once the kernels are finished the computed results have to be transferred back to host memory so that they can be returned to the user. Prior to this stage an initialization or binding against the GPU might be necessary. The details of these steps will be shown using CUDA as an example.

CUDA Runtime API The CUDA Runtime API offers semantic and syntactic extensions to the C programming language. It provides access to the GPU from a hybrid source file that contains both host and device code [NVI12a]. The main parts of a CUDA program are kernels and host based memory management for the GPU. The programming principle of a CUDA kernel is to write the steps to produce one output element in an array, just as the GPU generates the color information for one pixel on the screen. This is done in a fashion that also resembles the NVIDIA GPU processor architecture. An up to three-dimensional data set (termed "grid") is split into blocks of equal size. For every point/element within a block a thread will be created to compute the "result" for that point.[2] Thus, a very simple CUDA kernel looks like this:

```
// Mark function as 'global' so that it can be invoked from the host on the GPU
// Global functions always return 'void'
// Pre: array A of size N filled with valid values
// Post: elements of A are all incremented by 1

__global__ void foo (float *A, int N) {

   // who am I?
   // global index 'idx' = current block index * block size + thread index within block
   int idx=blockIdx.x*blockDim.x+threadIdx.x;

   // now just work on 'my' element
   // make sure that idx is not larger then N
   if (idx < N)  A[idx]=A[idx]+1.0f;
}
```

[2] The fixation of threads to data elements is usually done for practical purposes. It is, however possible to just create an array of threads without any correlation to data which is to be computed and then map the work to the threads in a semi-automatic fashion.

As noted before, the host is responsible for the memory management for itself as well as for the GPU. Hence, the CUDA Runtime API extends the traditional malloc/memcpy/free functionality on the host side with equivalent routines for the GPU memory (CUDAmalloc/CUDAmemcpy/CUDAfree). The main difficulty for a programmer using this concept is that the pointers for the two disjoint address spaces are both stored in host variables. Thus, a pointer intended for usage on the GPU can very well point to a valid memory location on the host—i.e. hold the same virtual address—and a wrong usage of a GPU pointer can result in chaos. A host program corresponding to the simple CUDA example shown before looks like this:

```
int main(void)
{
  float *a_host; // host data
  float *a_device; // device data
  int N = 1400; // array size
  int nBytes, i;  // temporary variables

  int blocksize=256; // number of elements in one block
  dim3 dimBlock(blocksize);

  // number of blocks to fit N elements in blocks of blocksize
  dim3 dimGrid(ceil(N/(float)blocksize));

  // allocate host data
  nBytes = N*sizeof(float);
  a_host = (float *)malloc(nBytes);

  // allocate space for a also on the device
  cudaMalloc((void **) &a_device, nBytes);

  // initialize elements in a
  for (i=0; i<N; i++) a_host[i] = 100.f + i;

  // copy data from the host memory to the device memory
  cudaMemcpy(a_device, a_host, nBytes, cudaMemcpyHostToDevice);

  // invoke 'foo' with the corresponding grid and blocksize
  foo<<<dimGrid,dimBlock>>>(a_device,N);

  // copy results back to host
  cudaMemcpy(a_host, a_device, nBytes, cudaMemcpyDeviceToHost);

  // use results

  // free allocated memory
  free(a_host);
  cudaFree(a_device);

  return 0;
}
```

The very simple nature of this approach compared to the hassles necessary before, which involved the "hijacking" of a graphics API and conversion of input and output data into and from image or texture data, appealed to a growing number of researchers even in the early stages of CUDA.

CUDA Driver API On a lower level the programmer also has access to the CUDA driver API [NVI12a]. Runtime API calls are automatically mapped by the runtime library onto driver API routines. The low level driver API, however, offers additional functionality. Furthermore, the program code for the host is free of language extensions and can be compiled by the standard host compiler instead of `nvcc`. On the other hand, the API is more complex, creates more verbose host programs and, thus, offers more chances to introduce hard-to-debug mistakes.

The main advantage of using this way of accessing NVIDIA GPUs is the ability to connect to multiple devices from a single thread. This would require multiple threads/processes with the runtime API. The CUDA driver API also has served as a model on how OpenCL accesses accelerator devices, as will be shown in a later section.

GPU Architecture

GPUs are massively multi-threaded devices with thousands of concurrent threads computing their independent outputs [WJMN09]. While there are differences in the architecture of the GPUs of the two remaining GPU vendors with a significant market impact and direct compute capabilites—AMD[3] and NVIDIA—a number of commonalities remain especially with respect to graphics memory management. Low-end graphics devices from both vendors use main host memory, high-end devices on the other hand utilize special graphics memory. Currently the fifth generation of GDDR memory delivers a much higher transfer bandwidth (currently up to 150 GB/s) at a much higher latency to the GPU compared to DDR main memory [PSAW09]. The desire for high bandwidth at the cost of latency is driven by the nature of the GPU: deliver high frame rates at very high resolutions for computer games. At full HD resolution (1920 times 1080 pixels) an image refresh rate of 100 Hz at 32bit color depth will produce about 830 MB/s of output data to be displayed. In order to generate that constant output stream, complex three-dimensional models of the scenery to be displayed need to be loaded and transformations need to be computed, thus, generating an even higher demand for input data bandwidth. The rather high memory access latency can lead to starvation of the GPU just as a CPU is stalling when it waits for data to be transferred from memory. This is, however, deemed tolerable since for the massively parallel nature of the GPU this will "just" result in one or a few pixels not updated for an amount of time beyond the human reception anyways.

In order to accomplish the demand for high data throughput and fast computation, all GPUs feature two common techniques:

- **Aggregation of compute cores to groups**
  Since computer graphics typically require the same operation to be executed on multiple, adjacent data, it heavily uses SIMD parallelism. As a result, a GPU can be best described as an array of loosely coupled SIMD units which in turn are a group of tightly coupled compute cores. Although GPU vendors describe GPUs as executing a very large number of individual threads, the smallest scheduling unit on a GPU is one operation executed by all cores in one group simultaneously on an array of these threads.
- **Small but very fast local memory for the core groups**
  The need for thread communication contradicts highly efficient massive multi-threading since communication easily introduces load imbalance and, thus, imposes an upper bound to the achievable speedup. GPUs circumvent this limitation by allowing thread synchronization only within the threads executed on one core group by means of a piece of memory which is local to the core group. Data in this local memory can be accessed with just one clock cycle latency. The technical

[3] ATI was procured by AMD in the year 2006 which continued to use the brand "ATI" for its graphics products. As of late 2010 all GPU products from AMD are only labeled as AMD devices and the brand ATI is not used any longer.

means to allow all threads in a thread array simultaneous access to the local memory are also used to synchronize these threads.

NVIDIA distinguishes three lines of GPU products: The gaming focused GeForce products, the Quadro series for professional graphics, and the Tesla series for GPU computing. While all three are capable of running CUDA and OpenCL applications, there are differences as described in more detail below. NVIDIA GPUs can be best described as an array of independent non-cache-coherent multi-core processors called streaming multiprocessors [NVI11b]. The number of computing cores per multiprocessor is always fixed for one chip generation while the total number of active multiprocessors as well as the clock speeds vary between the GeForce consumer products, the Quadro high-end graphic adapters and the Tesla compute accelerators. It corresponds to the number of threads that are executed as a SIMD array, called a warp. The current "Fermi" processors for example use 32 cores per multiprocessor while its predecessor and "Tesla" processors had 16.

AMD GPUs traditionally had a different architecture than NVIDIA GPUs. They employed VLIW parallelism to compute multiple elements of the same data point, e.g. dimensions of coordinates or color contributions. The earlier used five-way parallelism has been reduced in the previous "Cayman" architecture to four-way VLIW [Dem11]. In the latest GPU design named "Graphic Core Next (GCN)" the VLIW architecture is abandoned as a whole [MH11] and compute cores are much more similar to NVIDIA's. The AMD equivalent to NVIDIA's streaming multiprocessors is called a SIMD core, sometimes also SIMD engine, with 16 compute cores each. Operations are grouped in so called wavefronts that are assigned to one SIMD core, making it equivalent to NVIDIA's warps.

Furthermore, NVIDIA's high-end Fermi products (Quadro and Tesla) and the upcoming high-end GCN products offer features that make them especially appealing for long running simulations common to HPC:

- **ECC memory error correction for all levels of memory**
  The high clock speeds for the graphics memory and the GPU on consumer graphics adapters lead to an increase in bit errors and, thus, faulty computation [HP10]. One solution for this problem is to lower the clock frequency and in return also lower the likelihood of a bit error which is for example the case for the Tesla GPU computing products. Additionally, ECC memory error protection was introduced in the Fermi generation of NVIDIA GPUs and GCN generation of AMD GPUs. This offers the same level of memory error protection on the GPU as in standard CPU compute servers. ECC error protection is turned on by default in the Tesla series, selectable in the Quadro series and always disabled in the GeForce line. AMD has has announced ECC support in its latest FireStream GPUs.
- **Fast double precision computation**
  The NVIDIA Fermi GPU enables double precision floating point computation at half the speed of single precision computation. This is new to GPU computing where double precision support was half-heartily added by both NVIDIA and AMD since it is not required for "normal" graphics routines but necessary to attract more applications for GPU computing. Previous generations of GPUs had double precision performance limited to one eighth (NVIDIA) or one fifth (AMD) of the single precision performance. This reduced the proclaimed peak performance by such a drastic factor that a major impact for scientific simulations, which usually require double precision, was

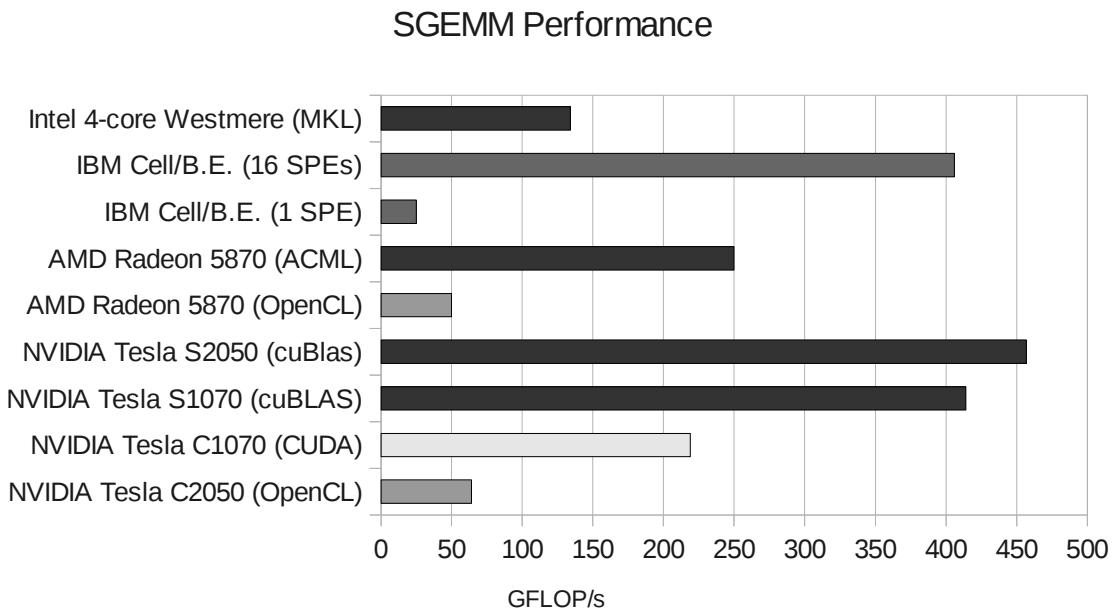

Figure 2.10: Comparing floating point compute performance of different accelerators using the native implementations or vendor provided libraries of the matrix-matrix-multiplication of single precision floating point numbers and 4096x4096 matrices.

still beyond reach. The Fermi GPUs now offer the same double precision "penalty" of a factor two as standard CPUs.

- **Fused multiply-add**
  The fused multiply-add instructions allow a higher computational throughput by combining frequently used floating point operations in one instruction, e.g. the common matrix-matrix multiplication makes heavy usage of this operation.

These features also mark a trend that GPUs are rather seen as device where a low power CPU offloads compute intensive tasks. Hence, GPUs are entering both the market for mobile devices as well as high-end servers and high performance computers.

AMD developed a new chip architecture called "Fusion"—now also called Accelerated Processing Unit (APU)—that combines the processing units of a GPU with a standard AMD x86 CPU in one processor. The APU-GPU part lacks the capability for computing with double precision floating point numbers, but in turn offers very fast on-die transfers for cached data. Furthermore, they only use standard DRAM as main memory which is targeted for low latencies and will not offer the bandwidth of typical SGRAM used for graphics adapters. AMD GPUs of any kind can be programmed with OpenCL which will be introduced in detail in the next section.

GPUs currently offer the fastest processing compared to other accelerators as shown in Figure 2.10. There are, however, a number of limitations to this fact: GPUs achieve far from peak performance for the SGEMM routine while the Cell processor and the Intel CPU achieve very close to their theoretic peak performance. Furthermore, the performance per Watt ratio is about the same for the GPUs and the CPUs.

2.3 Generic Accelerator Programming Paradigms

While some specific accelerator programming models like CUDA were very successful and also in principle designed to offer access for accelerator devices from different vendors, companies that built whole computer systems as well as programmers trying to leverage accelerator capabilities from a large number of computer systems are looking for more generic programming approaches. This has led to the OpenCL standard as well as multiple directive based approaches which are described in more detail in this section.

2.3.1 OpenCL

The Open Computing Language (OpenCL) defines a general framework for programming hardware accelerators [Khr11]. Current implementations exist for AMD and Intel x86 based CPUs, AMD and NVIDIA GPUs, Altera FPGAs, VIA VN1000 Digital Media Chips, but also the IBM Cell/B.E. and POWER based CPUs with an AltiVec extension. The OpenCL standard was proposed by Apple in 2008. The Khronos Group, which also hosts other standards like for example OpenGL, formed a working group which released the version 1.0 of the OpenCL standard in December 2008. Although current implementations only cover multi-core CPUs and GPUs, the standard is not limited to such architectures. A number of other vendors have joined the OpenCL consortium as listed in [Ros11].

Programming Model

OpenCL is based on the C99 standard of the C programming language and does not require a special compiler like for example CUDA albeit bearing close resemblance to it. Accelerator device and host processor are joined in a so called context, which enables accelerator access for the host program. Within this context an accelerator program is compiled and loaded by the driver on demand. The offloaded piece of work within this compiled program is also called a kernel (tagged as `__kernel`), and the concept of disjoint memory with the need for manual transfers is also in common. The execution of a kernel is started by enqueuing it into command queues that are associated to the context and device. The kernel is started with a number of global and local work items which distribute the work over the available hardware resources in a very similar manner to CUDA threads and thread blocks. OpenCL differentiates between in-order and out-of-order command queues depending on whether kernels in the queue can run concurrently (out-of-order) or not (in-order). It is, however, possible to have multiple in-order queues to generate parallel execution streams. Kernels can be associated with events, so that the start or completion of a kernel on the device can trigger the launch of another kernel from another queue and, thus, allow synchronization between the kernels.

Hardware Architecture

OpenCL itself is primarily a programming model, but also implies a certain hardware structure as shown in Figure 2.11. It consists of processing elements that are aggregated to compute units which in turn are aggregated to compute devices. This three level hierarchy of processing elements as well as memories has the flexibility to resemble current CPUs as well as accelerators like GPUs or the IBM Cell/B.E. as shown in Table 2.1.

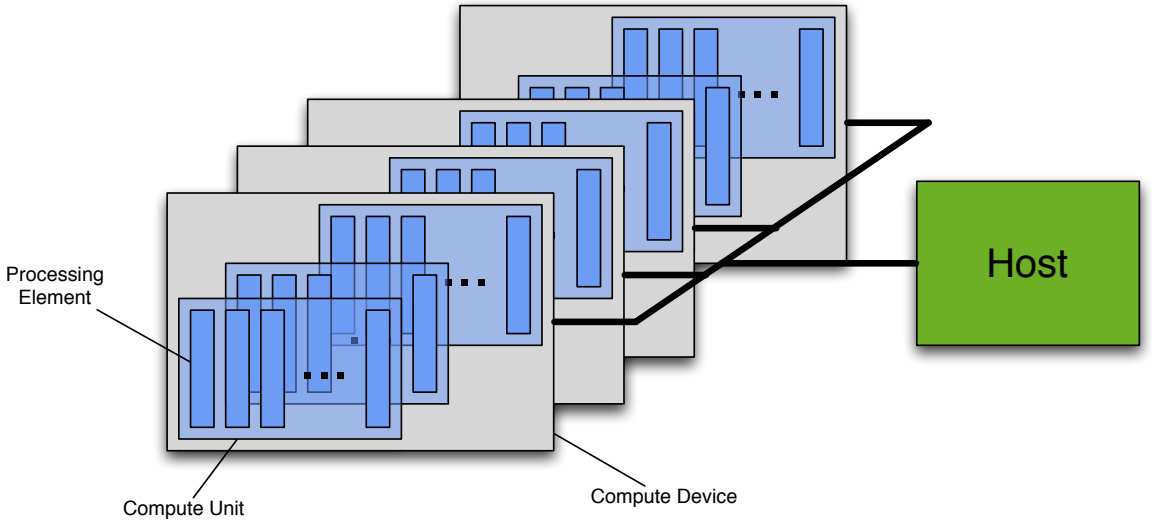

Figure 2.11: The OpenCL hardware model distinguishes between a host processor and a hierarchy of compute devices [Khr11].

Table 2.1: The OpenCL memory hierarchy distinguishes between three kinds of memory that can be mapped quite well to the kind of memory available in CPUs and GPUs.

Memory level	OpenCL term	IBM Cell/B.E.	NVIDIA GPU	Multicore processor
thread local	local variable	register	register	register
thread group shared	local memory	local memory	shared memory	shared cache
global memory	main memory	global memory	global memory	main memory

2.3.2 Directive Based Accelerator Programming

While CUDA and OpenCL require the programmer to write accelerator kernels himself as well as set up the work distribution, they bear a resemblance to native CPU thread programming, e.g., using pthreads. *OpenMP* was introduced as an abstraction targeted mainly, but not exclusively, for high performance computing to allow easy work distribution among threads without the need to know the fine details of thread programming [Ope11b]. It requires the programmer to introduce compiler directives (so called pragmas) into the application source code that explain how the following part of the application can be executed in parallel. The following example illustrates the principle:

```
#pragma omp parallel for
for (int i=0; i<MAX; i++)
   a[i]=b[i]+c[i];
```

The compiler takes this code sequence and generates a thread parallel version from it, that will distribute the iterations of the loop among the generated threads. Thus, data dependencies between loop iterations need to be avoided for this approach to work properly.

A similar programming model would be very beneficial for hardware accelerators as it would allow much easier code porting for existing applications already using OpenMP. A number of approaches to work towards this goal are presented in the following.

PGI Accelerator For C and Fortran based applications PGI Accelerator can automatically and transparently generate the necessary CUDA code segments for a number of code structures, for example loops [Por11]. Hence, a program segment using GPU acceleration could look like:

```
#define N 200000
...
#pragma acc region
for (int i=0; i<N; i++) {
   a[i]+=b[i]+c[i];
}
```

Its compilation results in:

```
juckel@joker:~/joker> pgcc -o foo foo.c -ta=nvidia -Minfo=accel -fast
main:
   16, Generating copyin(c[0:199999])
       Generating copyin(b[0:199999])
       Generating copy(a[0:199999])
       Generating compute capability 1.0 binary
       Generating compute capability 1.3 binary
       Generating compute capability 2.0 binary
   17, Loop is parallelizable
       Accelerator kernel generated
       17, #pragma acc for parallel, vector(256) /* blockIdx.x threadIdx.x */
           CC 1.0 : 5 registers; 44 shared, 4 constant, 0 local memory bytes; 100% occupancy
           CC 1.3 : 5 registers; 44 shared, 4 constant, 0 local memory bytes; 100% occupancy
           CC 2.0 : 10 registers; 4 shared, 56 constant, 0 local memory bytes; 100% occupancy
```

The compiler generates the routines for data transfer (output lines 3-5) as well as the accelerated code segment to be run on the GPU in multiple versions (lines 6-14) to suit a multitude of NVIDIA GPUs.

HMPP Developed by CAPS, HMPP (Hybrid Multicore Parallel Programming) is the name of a standard (OpenHMPP) as well as a compiler supporting this standard. The standard allows the injection of pragmas into the application source code which define regions (so called codelets) of the application that are to be executed on an accelerator. The compiler than generates different versions of the codelet using CUDA, OpenCL, MPI, or OpenMP. The invocation of the codelet will generate a call into the HMPP runtime environment which detects the available hardware platform and chooses the appropriate codelet to be executed. The benefit of this approach is clearly that the user only has to maintain one source code regardless of the target hardware to execute the application on. Nevertheless, every layer of abstraction also introduces limitations. HMPP for example lacks full C++ support.

OpenACC A very recent initiative tries to combine the benefits of PGI Accelerator and HMPP. At the Supercomputing Conference 2011 PGI, CAPS, Cray, and NVIDIA introduced OpenACC, a standard for pragma based accelerator programming [NVI11c, Ope11a]. While the implementers of the standard currently only support CUDA backends, it is possible to also generate OpenCL code. It is the aim of the group to fuse OpenACC with the currently ongoing efforts to include hardware accelerators into the next version of OpenMP.

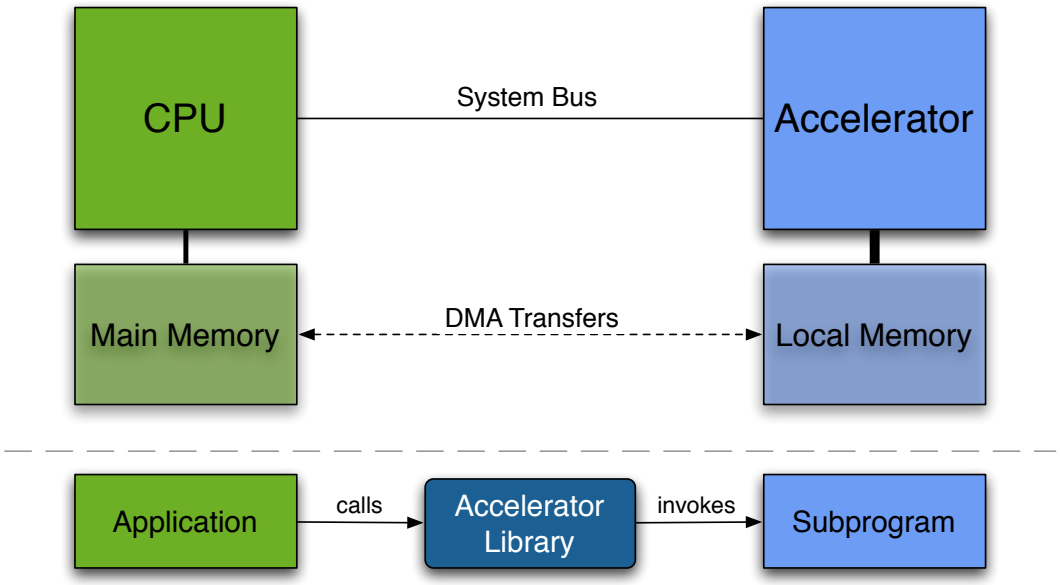

Figure 2.12: The typical setup for a system using a hardware accelerator connects the accelerator via a system bus to the CPU of the host system. Both the host system and the accelerator have their own memory, which requires data transfers over the system bus when the program running on the CPU wants to offload a subprogram onto the accelerator.

2.4 Common Charateristics of Hardware Accelerators

So far, the interaction with a typical hardware accelerator is purely host driven, i.e. the program running on the host CPU will offload work to the accelerator, wait for it to finish at some later point in time, and copy results back. The accelerator itself is typically not able to create new work by itself, it is used like a coprocessor in a different *ISA* domain. Access to the accelerator is provided via an accelerator specific driver and user space library, which offers an API for offloading work to the accelerator (see Figure 2.12).

While the presented accelerators are in itself quite different, they share the commonality of massively parallel data computation. Hence, a common goal in using any kind of accelerator is to exploit data parallelism in the application by locating data elements that can be independently and simultaneously updated. The offloaded work is typically put into special functions that the host can start on the accelerator device using the accelerator API. A frequently used term for the offloaded function is kernel, which will be used in this thesis from this point on. A kernel itself is basically a function pointer on the accelerator device which is inserted at the bottom of an execution queue. When the hardware scheduler on the device is looking for new work, it removes the topmost entry from an execution queue and start the corresponding device program on its computing elements. As already described in Section 2.3.1, two kinds of queues can be used: in-order and out-of-order queues depending on the order in which the entries are executed. Furthermore, a device can support multiple queues of the same kind, thus, allowing parallel kernel execution by executing the topmost kernel from each queue simultaneously. The principle of parallel execution queues is also referred to as execution streams.

The aforementioned parallelism in the kernels is typically distributed across at least two levels: SIMD

Table 2.2: The architectural characteristics of the various accelerators are quite different. Some require a very high SIMD parallelism while others require a lot of local data reuse to compensate a rather poor memory interface. [Klu11]

Accelerator	GFLOP/s per Watt	Peak GFLOP/s	FLOPs per double load	SIMD width
Xilinx Virtex 7 (28nm)	13.3	400	40	1
NVIDIA Kepler (28nm)	8.5	1,440	720	32
ClearSpeed e710 (90nm)	8.0	96	192	96
NVIDIA Fermi (40nm)	3.8	845	845	32
AMD Cayman (40nm)	3.4	1,270	635	16
Playstation3 (45nm)	2.1	100	38	2
PowerXCell 8i (65nm)	1.3	100	38	2
AMD Bulldozer (32nm)	1.3	147	24	2
Intel Westmere EX (32nm)	0.7	96	26	2

parallelism used within the computational units of the accelerator to compute multiple data elements simultaneously and domain decomposition so that several of the SIMD parallel domains can be computed concurrently by the parallel compute units of the device. This distinction can be translated to thread-parallel CPU programming as well. Table 2.2 lists performance characteristics for a number of current hardware accelerators as well as standard processors.

Another common feature of accelerators is the need for data transfers between different memory locations. In a typical setup both the host and the accelerator device have their own memory hierarchy, sometimes even using different technologies. Before the accelerator can start computing, it has to be supplied with the necessary input data, meaning that the host program has to invoke the accelerator API. The accelerator driver has to transfer data from host memory over the host bus—which is typically the *PCI-Express* bus—to the accelerator memory on the device. This can be done synchronously, i.e. the host program is waiting for the transfer to be completed, or asynchronously where the host program resumes immediately after offloading the data to the accelerator driver.

The memory hierarchies of accelerators have common features that are for example used by OpenCL to offer a device independent programming model. Accelerators feature a global memory which is usually smaller than the main memory of the host, a compute unit local memory, and a thread local memory. While the former is persistent over multiple kernel launches from within one context, i.e. one host application, and can, thus, be used for synchronization between different compute units, the latter two are cleared once the kernel for the thread or SIMD block is computed.

At this point it has to be noted again that the IBM Cell/B.E. is a special kind of accelerator as it combines features of standard CPUs and accelerators. It offers a global memory for both the host processor (PPE) and the accelerator (SPEs). Therefore, it does not require dedicated data transfers between them. Nevertheless, the Cell SPEs also feature their local memory which has to be filled by the SPEs themselves much like a GPU program uses its local memory.

The upcoming Intel Xeon Phi implementation of the MIC architecture is also a slightly different form of hardware accelerator. It can be best described as a highly parallel in-order x86 processor residing inside the host system. The MIC can be used twofold: as an offload engine for the host or as a stand-alone component executing own work. In the former case it follows the same generic principles established in

Table 2.3: Comparison of architectural features for the IBM Cell/B.E., the NVIDIA GF100 (Fermi) GPU, and a general purpose multi-core CPU [HJB12]

Feature	IBM Cell/B.E.	NVIDIA GPU	Multi-core CPU
Host processor	PPE	CPU	one core
Accelerator	SPE	GPU	all cores
Local memory	256 KB local store	16-48 KB shared memory	local cache (L1/L2)
Device memory	–	memory on the graphics board	shared Cache (L3)
Host memory	main memory (EIB attached)	main memory of the host system	main memory
Data transfer	DMA transfers	CUDA memcopy	shared address space
Synchronization	mailbox messages	CUDA thread synchronization	shared address space
Accelerated program	SPE program	CUDA kernel	multi-threaded program

this section. In the latter case it can be treated like an own host, and the existing performance monitoring approaches for multithreaded applications can be applied.

Table 2.3 lists the common features of some accelerators and introduces common terms that will be used in this thesis whenever addressing general accelerator features. Wherever possible OpenCL terms are used since this is the programming model with the largest distribution among all available hardware accelerators.

After introducing multiple hardware accelerators and their programming models, the next chapter will discuss how performance of applications in general can be monitored and how various performance tools implement these methods.

3 State-of-the-Art in Application Performance Analysis and Related Work

Performance analysis for any kind of application requires three steps: data acquisition or monitoring, data recording, and data presentation [Jai91]. Within these steps different techniques can be used as shown in Figure 3.1. Unfortunately, the names of these techniques are used ambiguously. This thesis uses the terms introduced by JAIN and extends them to also include more recent techniques and terms.

A running program can be monitored in two different ways. One is a pull-method (sampling) that interrupts the program at arbitrary points in time and notes the state of the observed program. The other method is a push-method (event triggered) where the running program or execution environment is modified in such a way that it invokes the monitoring tool at specific points during its execution to inform the monitor about an updated program state. It is obvious that the former offers an almost constant overhead in the program execution time while at the same time having the accuracy of the measurement being highly dependent on the sampling frequency. For the latter it is vice-versa, since the rate at which events occur varies between different programs. The perturbation of the monitored application—in imitation of JAIN's proposed system-under-test for monitored systems hence called "program-under-test"—is unknown, but no performance relevant event will be missed.

After the state of the program-under-test is known the performance monitor has to pass this information to a data presentation program. This can be achieved by just passing a memory data structure if the presentation is a part of the monitoring tool. Nevertheless, data recording is typically done in some kind of file output. The output method but also the internal memory structures of the performance monitor can again be twofold. It can on the one hand be aggregated so that only a summary is recorded. As events or samples are monitored their data is immediately fused with the previously recorded data. This method allows for a very small memory footprint of the performance monitor, but also sacrifices detail. The performance analyst receives a report about performance relevant activity but has no understanding at which point in time during the program execution that activity occurred. Event logging or tracing, on the other hand, records each monitored program state together with a time stamp so that an exact replay of the program execution is possible. On the downside, logging requires a much larger amount of memory for the monitor to store all these records which the system-under-test must have available. It is, however, possible to limit the amount of required memory by flushing the memory buffers to disk whenever they exceed a certain threshold.

The recorded performance data can be presented in two different ways: as performance profiles or as timelines. A performance profile is a table or chart displaying the distribution of one or more performance metrics over the observed program's parts or processes. A typical example is how the overall program run time is distributed over the various functions of the program or how often each function is invoked. Timeline-based visualization on the contrary reconstructs the initially recorded program states along an

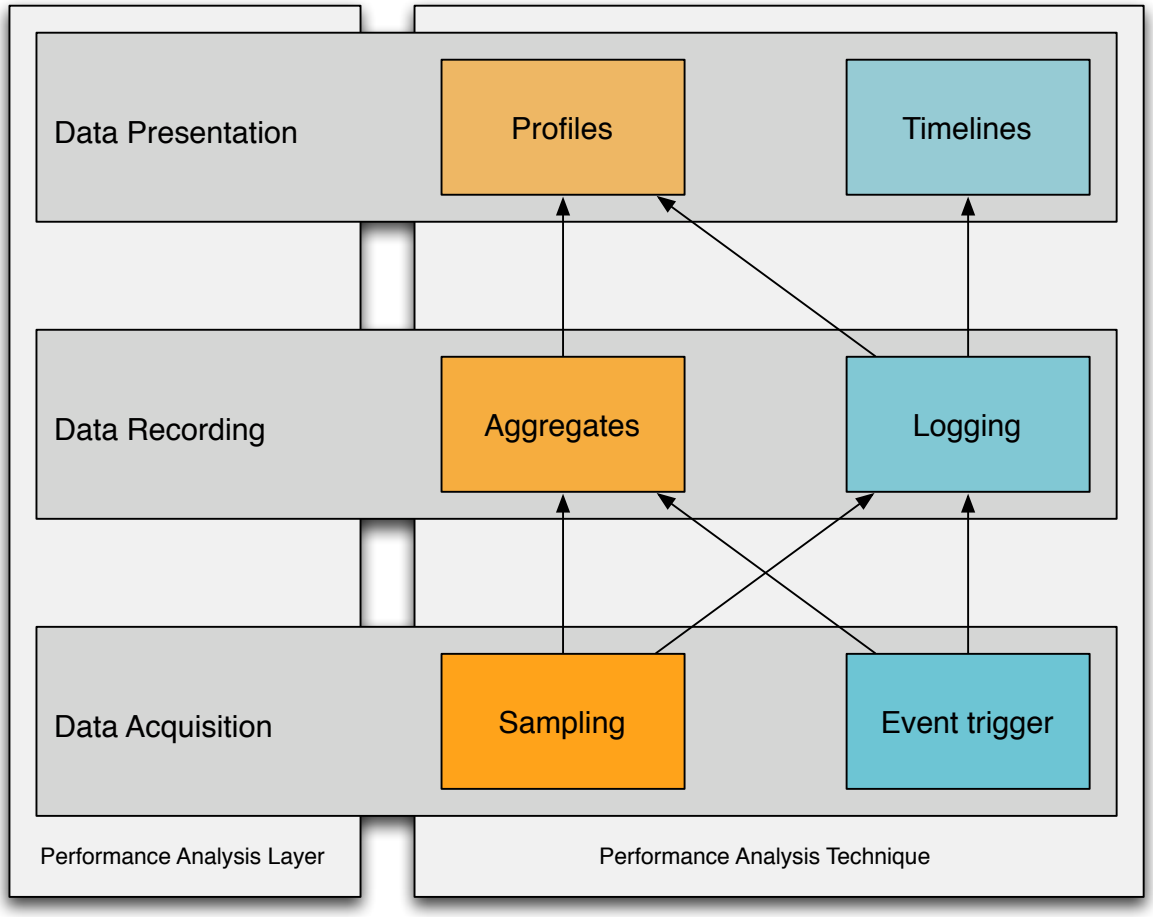

Figure 3.1: Performance analysis is a three staged process of acquisition, recording, and presentation. Each step can employ different techniques which can be exchanged. The only limitation is that aggregated data cannot be expanded again to produce a timeline presentation.

increasing time axis, thus, exactly showing when the program did what. This method of illustration helps locating varying behavior of the program-under-test over multiple phases of the program execution. It is possible to generate both profiles and timelines from trace data while the reconstruction of a timeline from aggregated data is impossible and only allows performance profiles. The term “profiling”, as it is used very frequently to describe performance analysis in general, actually refers to generating profiles of aggregated data that were either gathered by sampling or triggered by events.

This chapter explains techniques to implement trace based event logging in more detail and discusses multiple presentation techniques for data from event logs. It, furthermore, provides an overview of existing performance analysis tools that also cover hardware accelerator activity.

3.1 Gathering Performance Data Using Event Logs

Event logging is a method to acquire and record performance related data during the execution of the program-under-test. In order to gather this data, the measurement environment is fused with the program

such that the program interrupts its execution to invoke the monitoring environment to note the occurrence of a certain event. Thus, a state change in the observed program will trigger an event which can then be recorded. A simple code example will illustrate the basic principle. To record the time spent for executing a function `foo`, an event needs to be triggered when entering and leaving the function.

```
void foo(){

  // do something

  return;
}
```

⇒

```
#include <trigger.h>

void foo(){
  trigger_enter("foo");

  // do something

  trigger_leave("foo");
  return;
}
```

The handling of the event triggers is part of the measurement library whose handles are available to `foo` after including `trigger.h`. The corresponding handling routines in the monitor then have to generate a time stamp and record the type of event with its properties.

```
#include <trigger.h>

buffer events[SIZE];
int current_position;

void trigger_enter(char *name){
  double timestamp=get_high_resolution_time();
  sprintf(events[current_position],"%g: Enter %s",timestamp,name);
  current_position++;
  return;
}

void trigger_leave(char *name){
  double timestamp=get_high_resolution_time();
  sprintf(events[current_position],"%g: Leave %s",timestamp,name);
  current_position++;
  return;
}
```

An event based performance monitor is typically implemented as a system library that can be linked with the program-under-test [Ski05, MKJ+08, SM06]. The monitor implements handlers for the various types of events that can be triggered. Possible events are entering and leaving of arbitrary code regions of interest, creation or destruction of additional threads to work on the program concurrently, communication between threads or cooperating processes, or the invocation of system libraries. The process of modifying the program-under-test to trigger the events is called instrumentation. Depending on how these triggers are inserted into the program-under-test, one differentiates three different types of instrumentation. The instrumentation methods are explained using VampirTrace (see Section 3.5.1) as an example, since this is currently the best scaling event logging performance monitor.

Manual Instrumentation

Manual instrumentation requires the performance analyst to add the event triggers into the source code of the program-under-test, recompile the program, link it with the performance monitor, and execute it. In order to trigger the events correctly, the performance monitor API needs to be available to the performance analyst and well documented, so that the analyst can add the right triggers to the program-under-test. Manual instrumentation for the tool VampirTrace is done via preprocessor macros that are

available after including the `vt_user.h` header file. The VampirTrace user API offers only routines to record the begin and end of code sequences, to insert custom markers into the event stream, to turn the event logging on and off, and to manage event buffers. There are, however, also low level APIs to include other information such as hardware performance counter data or source code location information. It is the performance analyst's responsibility to ensure that each "begin"-trigger is matched by a corresponding "end"-trigger, hence, if a code sequence has multiple exit points each of them has to be instrumented. Failing to do so will result in an incomplete performance analysis, since program profiles or timelines will be computed by replaying the events and feeding them into a state machine. This will not work for unbalanced begin and end log entries. While this approach places a high burden on the analyst to ensure correctness of the measurements, it also allows the highest resolution for the analysis, since any code region can be instrumented. Therefore, manual instrumentation is the option for recording events on a basic block level.

The following code listing shows how a user can instrument a loop:

```
#include <vt_user.h>

void foo();

int main(int argc, char **argv){
  VT_OFF(); // turn event logging off

  // do some stuff

  foo();

  // do more stuff

  VT_ON(); // turn event logging back on
  return 0;
}

void foo(){
  int i;

  // do initial work

  VT_ON(); // turn event logging on again
  for(i=0;i<10;i++){
    VT_USER_START("loop"); // record begin of loop iteration

    // do work

    VT_USER_END("loop"); // record end of loop iteration
  }
  VT_OFF();

  // do more

  return;

}
```

Compiler Based Instrumentation

Typical points for event triggers are entering and leaving of functions in the program-under-test. Generating triggers for leaving a subroutine is complicated, since it has to be done for every exit point of the function. The following code listing illustrates the principle using the VampirTrace User API:

```
#include <vt_user.h>

void foo();

int main(int argc, char **argv){
  VT_USER_Start("main"); // record begin of "main"

  // do some stuff

  foo();

  // do more stuff

  VT_USER_END("main"); // record end of "main"
  return 0;
}

void foo(){
  int i;

  VT_USER_START("foo"); // record begin of "foo"

  // do some stuff

  if (i==42){
    VT_USER_END("foo"); // early end of "foo" on special condition
    return;
  }

  // do more

  VT_USER_END("foo"); // record normal end of "foo"
  return;

}
```

If the whole program-under-test were to be fully instrumented, these steps would be repeated for all functions of the program. This tedious task can be aided by the compiler that is used to translate the program to an executable. Compiler instrumentation requests the compiler to instrument every function entry and exit point. This results in calls to an event trigger handler being inserted at the appropriate locations in the program. The event trigger handler has to be implemented by the performance monitor. In case of the GNU compiler it receives the memory address of the invoked function as an argument [Fre11]. The memory address can be turned into a human readable function name by looking it up in the symbol table of the program-under-test. This look-up can happen for every triggered event, or after the monitoring run of the program. The latter requires the symbol table to be recorded as well, so that the address-to-name translation is possible. It is obvious that the instant look-up introduces a higher program perturbation, since it extends the time the program-under-test is interrupted by the event trigger handler. Thus, a post mortem translation either at the end of or after the monitoring run is preferable.

The already mentioned GNU compiler requires the following two functions to be implemented by the performance monitor [Fre11]:

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit  (void *this_fn, void *call_site);
```

If the program-under-test is compiled using the option `-finstrument-functions`, the compiler will insert these two calls at every begin of a function and at all exit points. The first argument of the function call is the address of the function whose entry or exit is observed. This address needs to be translated using the application's symbol table to the original function name.

While compiler instrumentation is at first sight much easier for the performance analyst, it can lead to a number of undesired side effects. First of all it can create "ghost" functions. These are subroutines that the compiler has instrumented, but which have been *inlined* in a later optimization step by the compiler. Thus, the even trigger handler in current compiler implementations receives the address of a parent of the actually triggered function which wrongly suggests a recursive function call. The only way to circumvent this problem is to disable function *inlining* for the compilation of the program. This does, however, change the execution time of the program in a non-deterministic way, since it will introduce additional function calls.

Secondly, the overhead for running the instrumented program is almost impossible to estimate without a prior profiling run. Since each handling of an event trigger takes a certain amount of time which can be considered constant, it is obvious that the program perturbation is proportional to the event frequency. A prior profiling run of the program-under-test can derive a lower bound for the number of function invocations and an upper bound for the program runtime. Thus, knowing the duration of handling one event trigger, the overall measurement overhead can be closely approximated. This is especially important for applications using object-oriented programming, since they typically consist of a lot of very short subroutines. In order to limit the overhead, the compilation process can be modified so that only certain source code files are compiled using compiler instrumentation. Furthermore, filter lists can be passed to the compiler for functions that are to be excluded from the instrumentation.

Lastly, compiler instrumentation, of course, requires a compiler that can actually do this instrumentation. While a wide range of compilers used for computational science on standard processors support instrumentation, special compilers typically used for hardware accelerators lack this functionality, especially for the accelerated code sections. VampirTrace can work with a large number of compilers that support instrumentation: GNU, Intel, PathScale, Portland Group, IBM, NEC SX, OpenUH.

Binary Instrumentation

In addition to the previously presented monitoring methods, it is possible to modify the compiled program-under-test using binary instrumentation. With this method the compiled machine code is modified such that a function call and the return from a function invokes a special method that is added by the monitor to the program-under-test. The added functions generates event triggers which can than be handled by the monitor in the same way as using the other previously introduced instrumentation methods.

The challenge with binary instrumentation is its platform dependency. The monitor employing this technique has to be able to disassemble the program-under-test to locate the operations that should trigger an event. This results in a very close dependency to the *ISA* that is used by the program. Since most *ISA*s evolve by adding new operations, the monitor has also to be kept up to date, to cope with the new instructions. Dyninst is an example for a tool that supports binary instrumentation [HB00].

Library Interception

Monitored parallel programs regularly utilize a number of system libraries. From a performance point of view, the libraries for communication among processes and threads are of largest interest since their usage

directly influences the overall program performance. This can be achieved using special tool interfaces provided by the libraries. A popular example is the profiling interface of the MPI library [Mes95]. In addition to the "normal" way to invoke routines of the MPI library with their `MPI_*`-Names, the profiling interface asks for them to be also available via `PMPI_*`. The original `MPI_*`-Routines are implemented as weak symbols, thus, allowing performance monitors to substitute them. As a result the call to the MPI routine by the program-under-test will be directed to the performance monitor that can then handle the event and invoke the MPI library using the `PMPI_*` call.

Threading libraries unfortunately do not offer such a profiling interface. Nevertheless, if the program-under-test is instrumented by any of the three presented means and uses multiple threads, the performance monitor can capture the thread utilization. Threading libraries, like the widely used pthreads, launch new threads by starting a subroutine of the program. The instrumentation of the program causes an event to be triggered as a subroutine is started, thus, invoking the performance monitor to handle the event. The performance monitor can then use thread identification routines supplied by the threading library to determine which thread is executing the called subroutine. For a whole program run the performance monitor can record the whole thread usage of the program-under-test.

Both approaches have the shortcoming that either the library or the program-under-test has to generate events that are than handled by the performance monitor. A more generic way is desirable and is presented as a first step in this thesis in Section 4.1.

Limitations

While event logging can provide the performance analyst with the complete picture of what the program-under-test did at any point in time during its execution, the technique also has limitations. One concern is program perturbation. The more often events occur, the higher the perturbation and, therefore, also the execution time of the program-under-test. Especially for event triggered performance monitors the overhead in execution time is unknown prior to the measurement. The data acquired from a highly perturbed program-under-test is semantically still correct, meaning that the sequence of events remains the same. Any quantitative information, however, has to be considered with great caution, because potential load imbalances or long communication times can actually be induced by the performance monitoring overhead. As a result the execution time overhead has to be considered as well as part of the performance analysis that uses the event log as well. Another possibility is to use sample based event logging which provides a constant overhead due to the well defined intervals for test samples. Here the sampling frequency determines the accuracy of the recorded data—a low sampling rate can result in missed program states that are of interest. This problem and potential solutions, however, are not new and have for example been discussed in [Sha49] or [SH11].

Another challenge is handling the data volume generated by the performance monitor. A recorded event typically consists at least of a timestamp (8 bytes), an event type (4 byte), a process or thread number (4 bytes), and an identifier for the subroutine the event was triggered by (4 bytes). If one assumes an event rate of one event every 900 clock ticks for a program running on a processor clocked at 1 GHz where the event handling will take 100 clock ticks, this setup will generate 1 million events per second or a data stream of 20 MB/s. For a sequential program running 100 s this results in about 2 GB of event data.

Keeping this data in main memory might compete with the memory requirements of the program-under-test and, thus, be impractical. As a result, the event log data has to be transferred to an I/O device at some points during the measurement which will result in an even larger overhead. This I/O problem as well as the problem of event data sizes becomes even more eminent if parallel program execution comes into play. Parallel programs on very large supercomputers can nowadays be comprised of 200,000 or more processes, resulting for the mentioned example in 400 TB of event data or an aggregated data stream of 4 TB/s. At this point event logging is impractical even in HPC environments. A potential solution to reduce the amount of data is logging only a limited number of preselected events. A more sophisticated approach [MK12] filters during the execution of the program-under-test, but this filtering decision also induces a higher measurement overhead.

3.2 Performance Profiles and Timelines

As already mentioned the introduction of this chapter, the kind of possible performance data presentation depends on how the performance data was recorded. If it was immediately aggregated only these aggregates can be presented. Typical performance metrics of that kind are number of invocations and time spent in the various subroutines of the program-under-test. The simplest form of presenting the aggregates for those two metrics is a table listing the subroutine names and the recorded metric. This tabular data can be turned into different charts to show how it is distributed. Typical charts for that matter are bar or pie charts.

The same performance profiles can also be generated from event logs. In this case the data has to be aggregated from the recorded events by the data presentation tool. In addition to profiles an event log can be used to present the temporal order of events which is called a timeline chart. In it the program states—which subroutine was running at every point in time during the exection of the program-under-test—are derived by using a state machine which reconstructs the call stack of the program-under-test and serializes it for a flat presentation. An example for all three forms of presenting performance data of a sequential program is shown in Figure 3.2.

If the program-under-test is a parallel application using multiple processes or threads, a performance profile will be able to present the same kind of information as an aggregate for all participating parts of the program. As a first possibility the individual statistics of all processes/threads of the program-under-test can be aggregated to provide an overall execution time distribution or invocations count for all recorded subroutines. Doing so provides the performance analyst with an answer to a number of questions regarding the performance of the program under test:

- In which subroutine is most of the execution time spend? If it is not a system library, e.g. a communication subroutine, then this is a good candidate for intra-procedural optimization.
- Are calls to the parallelization library using a lot of time? If this is the case, the program suffers from load imbalances or bad communication patterns. A more detailed study of various parts of the application with an event based performance tool will reveal those bad performance patterns.
- Are the subroutines that are called most often not using a lot of execution time? These are candidates for inlining to reduce the overhead induced by subroutine calls.

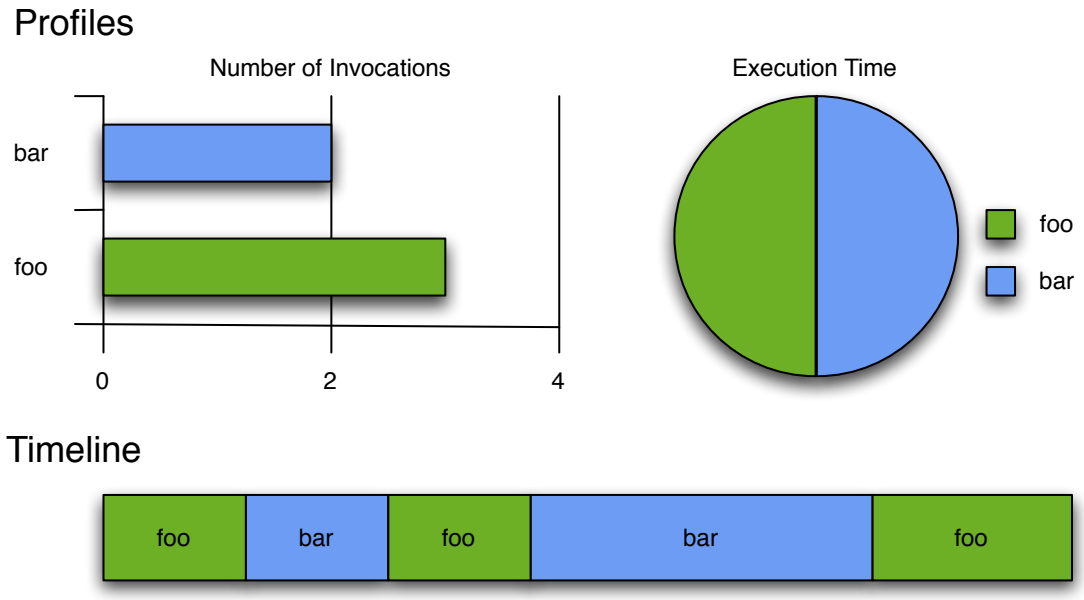

Figure 3.2: Event log data can be visualized by statistical profiles or a timeline representation of the event log. The profiles are generated by aggregating the event log data post mortem, just as a aggregate recording tool would do online.

Some profile visualization tools like for example TAU (introduced in more detail in Section 3.5.2) also allow the presentation of differences between the profiles of the individual processes/threads of a parallel application. Such a visualization can fuse the data of threads or processes that behave similarly and present the processes/threads with a different behavior. This can be anticipated in case of a master thread or process but could also indicate the cause for a load imbalance such that some processes/threads have more or less work to do then others and, thus, spend less or more time in the communication library.

Parallel timeline visualization of event logs from parallel applications on the other hand can present the whole temporal execution flow of the program-under-test. In this case the execution patterns of the individual processes or threads become immediately visible as the mutliple timelines are placed right next to one another. Figure 3.3 shows how a load imbalance can be easily detected by a performance analyst using this kind of display for the event log data. Displaying timelines of very long running applications will lead to aliasing effects as the number of events easily surpasses the number of pixels in the horizontal direction on the screen to display the information. As a result the presentation tool has to decide which event or program state takes precedence over the others and shall be displayed. The decision is made such that the ratio of pixels colored in the various colors of the corresponding function groups reflect the ration of the execution time of the function groups [Bru10]. The same is also true if the number of parallel event streams surpasses the number of pixels in the vertical direction of the screen. It is a logical course of action to be able to limit the displayed portion of the event log in the temporal and spatial (number of event streams) dimension to an excerpt from the event log, thus, being able to display more data that was previously hidden. At this point, the profile charts that have been generated from the event log data can be updated as well to reflect only the currently displayed excerpt. This allows a very interactive way of studying the performance characteristics of the program-under-test and can also cope

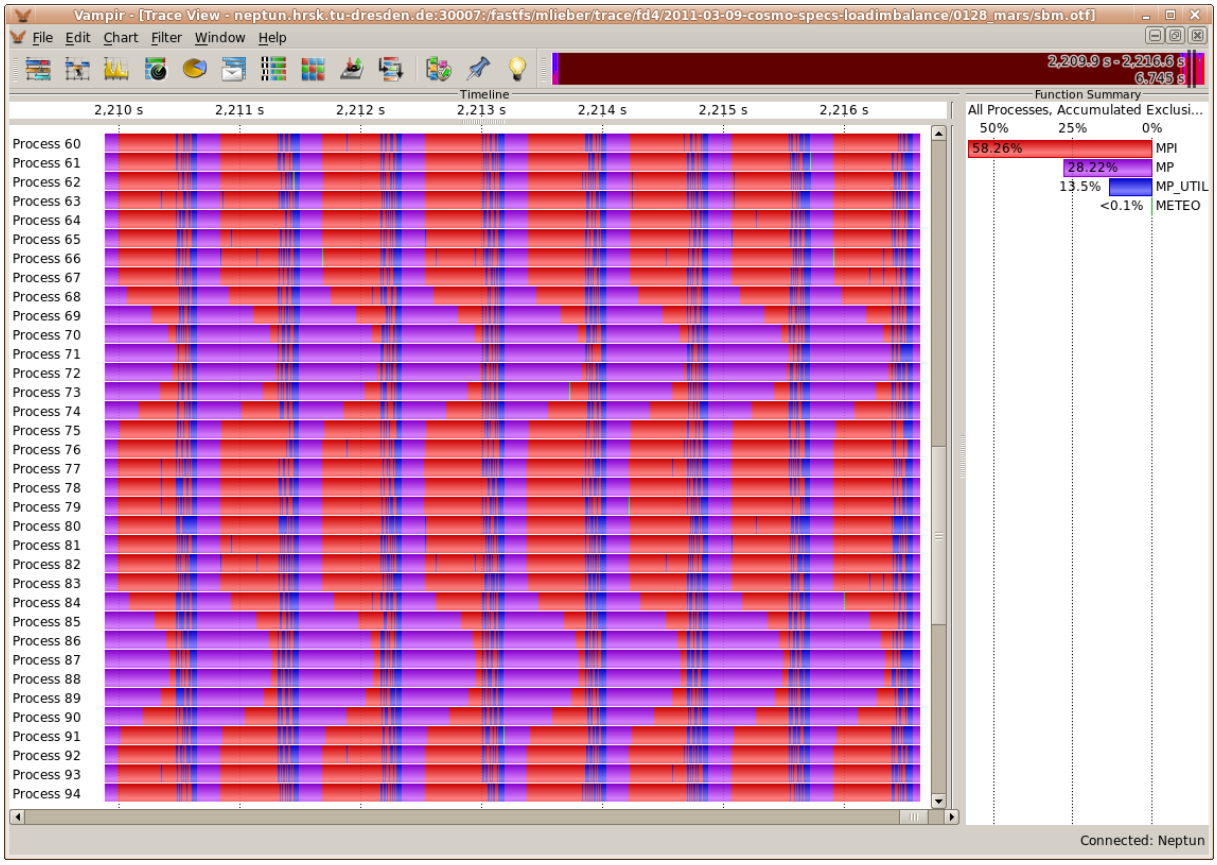

Figure 3.3: The timeline representation of an event log easily shows a load imbalance in the program-under-test. In this case a coupled wheather-multiphysics application experiences a difference in runtime for the multiphyiscs phase depending on the MPI rank. This is due to an increased load when the multiphysics code has to simulate cloud activiy. [Lie12]

with changing behavior of the application that could be smudged in the aggregates generated by a pure profiling tool.

Another strength of an event triggered trace based performance analysis for parallel applications is the capability to capture the arguments passed to the used communication library. This can be used to generate a number of profiles on the communication patterns like message size distribution or a communication matrix as shown in Figure 3.4.

It should also be obvious that the task event log analysis and presentation tool is much more complex than presenting charts of already generated aggregates. As a result trace analyzers became also parallel applications to be able to cope with the large amounts of event log data generated by highly parallel programs-under-test [Bru07].

3.3 Repeating Patterns in Event Logs

KNUEPFER introduced in [Knü08] Complete Call Graphs (CCG) as a novel memory data structure to represent an event log. The nodes in the graph represent program states or subroutines, the edges transitions from one state to another. Multiple event streams of one event log are represented by multiple

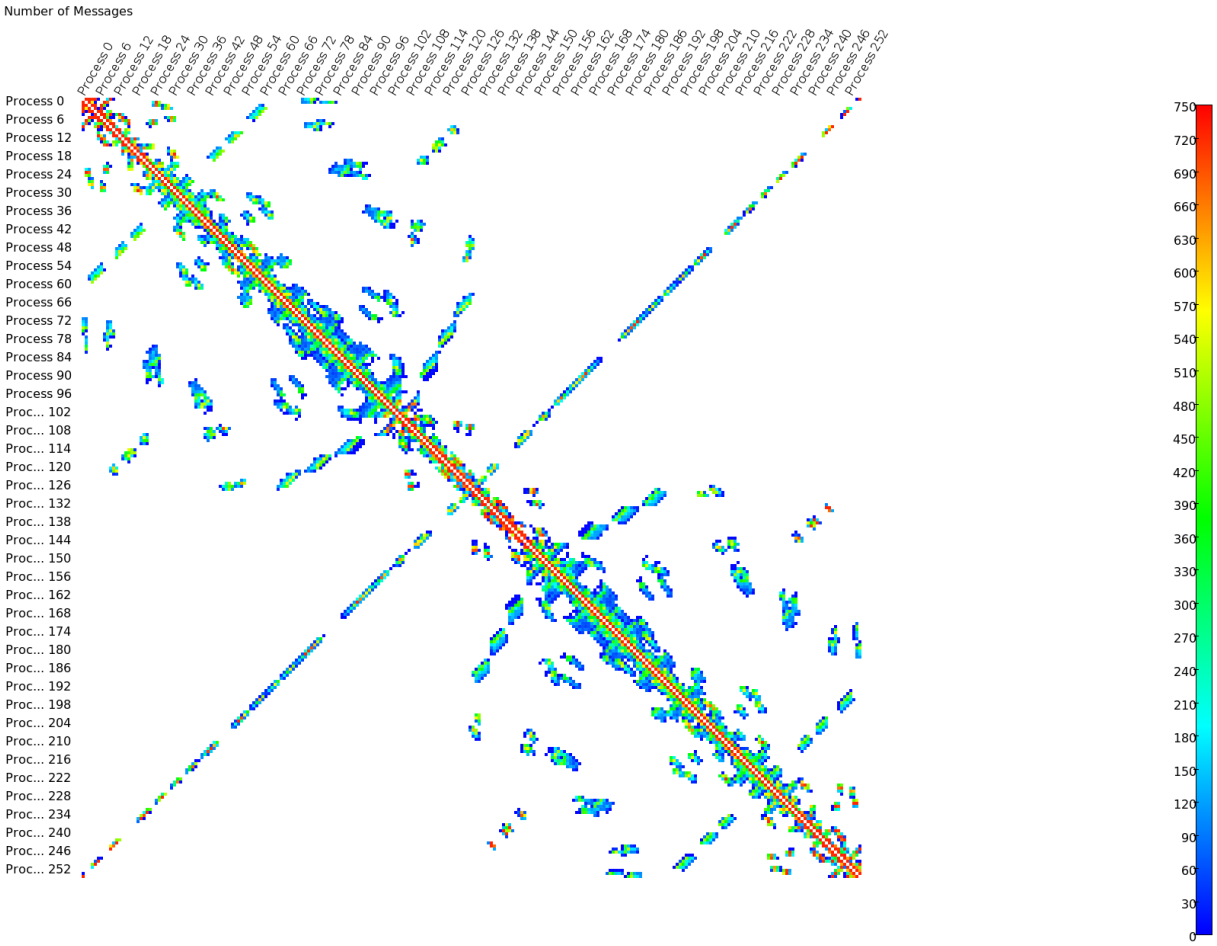

Figure 3.4: The communication matrix shows MPI message properties, such as number of messages, message sizes, or transmission rates, with respect to the communcation partners. In this case most communication happens between neighboring MPI ranks. [Lie12]

graphs. A node of the graph has hard and soft properties such as a unique state identifier and a state duration.

A CCG first used the idea to use only durations together with events which proved especially useful when the idea of the Compressed CCG (C3G) was presented. The compression is based on matching nodes with the same hard properties by allowing some deviation in the soft properties. Applied to real event logs this will lead to a subroutine being called multiple times with a duration that is within the defined limits to be only represented by one node of the graph, thus, aggregating similar program states. This principle can be extended to multiple graphs as well. In this case the similarities are also searched from multiple graphs which fuses graphs when they are found.

In order to support typical operations for performance data presentation like providing a timeline display or generating a performance profile, the C3G can be traversed in correct temporal order. The same linearization is necessary to store the graph in one of the standard event log file formats. One specialty of C3Gs is that they have a maximum number of children per node. This is achieved inserting dummy nodes in case the number of children exceed the maximum and rebalancing the subtree. This graph based event log representation will later on serve as a basis for a novel event log presentation approach.

WEBER ET.AL. introduced in [WBB12] an approach to compare event logs from multiple executions of the the application using a hierarchical genome sequence alignment approach. In this approach two event streams from two different event logs are compared by generating a call sequence and aligning this sequence. Due to the large number of program states and the quadratic complexity of the alignment algorithm, the authors propose to statically split the sequence using the call stack level and a global maximum.

3.4 Existing Performance Tools from Hardware Accelerator Vendors

Accelerator vendors typically offers some kind of performance analysis tool as part of the programming kit for the accelerator. They are usually sample based profiling tools which generate statistics about how the accelerator is used. In contrast to the vendor supplied tools are third party performance analysis tools which are typically developed in a research environment and try to be platform independent. A number of these tools has also embraced hardware accelerators and will be introduced in a later section of this chapter as well.

A commonalty among vendor tools is their limitation to study a program using the accelerator from the vendor. Nevertheless, they usually enable the most detailed observations of the accelerated program since the vendor tools utilize internal, non-public APIs. Therefore, they are typically published using a proprietary license. Tools from various accelerator vendors, as discussed in Chapter 2, are introduced in the following.

3.4.1 IBM Cell Broadband Engine

The Cell/B.E. software development kit (SDK) comes with a number of performance tools to ease programming of well performing applications on that platform [Hab08]. Aside from the profiling tool "OProfile" the SDK also contains the Cell Performance Counter tool (CPC) which enables access to the hardware counters of the processor. OProfile can also use CPC to provide a very detailed, sample based view into the execution of Cell/B.E. applications.

The Cell/B.E. SDK also features a tracing tool called Performance Debugging Tool (PDT). It instruments SPE and PPE code to record performance relevant events during program execution. It comes with a trace reader and summary generator for trace postprocessing. The PDT infrastructure for recording trace information requires that all trace data is sent to the PPE for recording, thus, limiting the scalability of this tool especially for 16 SPE configurations.

The most versatile tool of the Cell/B.E. SDK is the Visual Performance Analyzer (VPA), a plugin for the Eclipse IDE that consists of

- a profile analyzer similar to OProfile,
- a code analyzer to study basic blocks, functions, and assembly instructions,
- a pipeline analyzer,

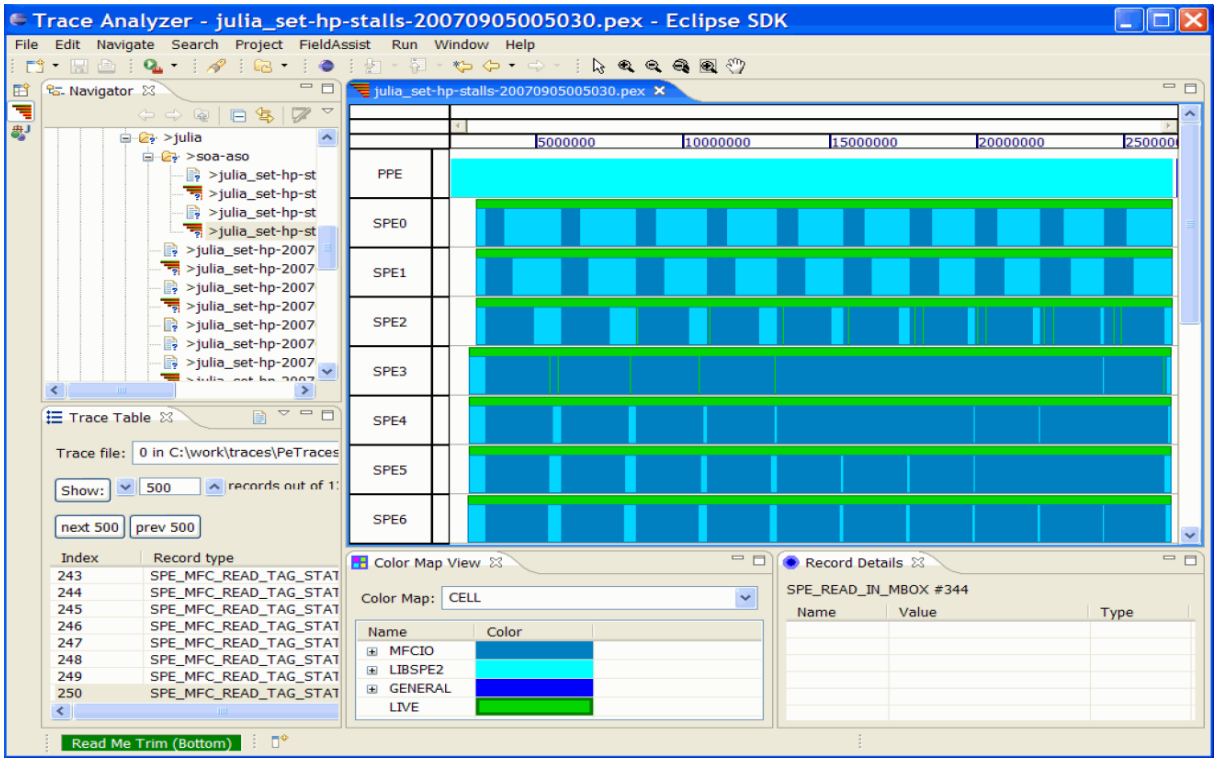

Figure 3.5: The Visual Performance Analyzer for the Cell/B.E. can be used to display event log data from the Performance Debugging Tool. The main display shows a timeline representation of the PPE and SPE activity. [Hac08]

- a counter analyzer for access to hardware counters,
- a trace analyzer to visualize trace information, and
- a control flow analyzer which also reads trace data.

The visualization of a PDT trace with Visual Performance Analyzer is shown in Figure 3.5.

3.4.2 ClearSpeed Accelerators

The ClearSpeed SDK features a performance tool called “Visual Profiler” [Cle10]. Although the name suggests otherwise, it is the visualization frontend for trace and sample based data and features a timeline display to show precise execution information of ClearSpeed accelerated applications. The event recording infrastructure is part of the ClearSpeed execution environment and can be enabled by simply setting the following environment variables:

```
CS_CSAPI_TRACE=1
CS_CSAPI_TRACE_CSVPROF=1
CS_CSAPI_TRACE_CSVPROF_FILE="<outputfile>.cst"
```

The trace information is then captured directly on the device by special hardware, processed by the supplied debugger “csgdb” into the text format of the cst-traces. This trace file can be opened in the Visual Profiler as shown in Figure 3.6. The tool allows a very detailed study of the accelerated application by aligning the source code with the performance data and also displaying detailed pipeline information captured from the CSX processor.

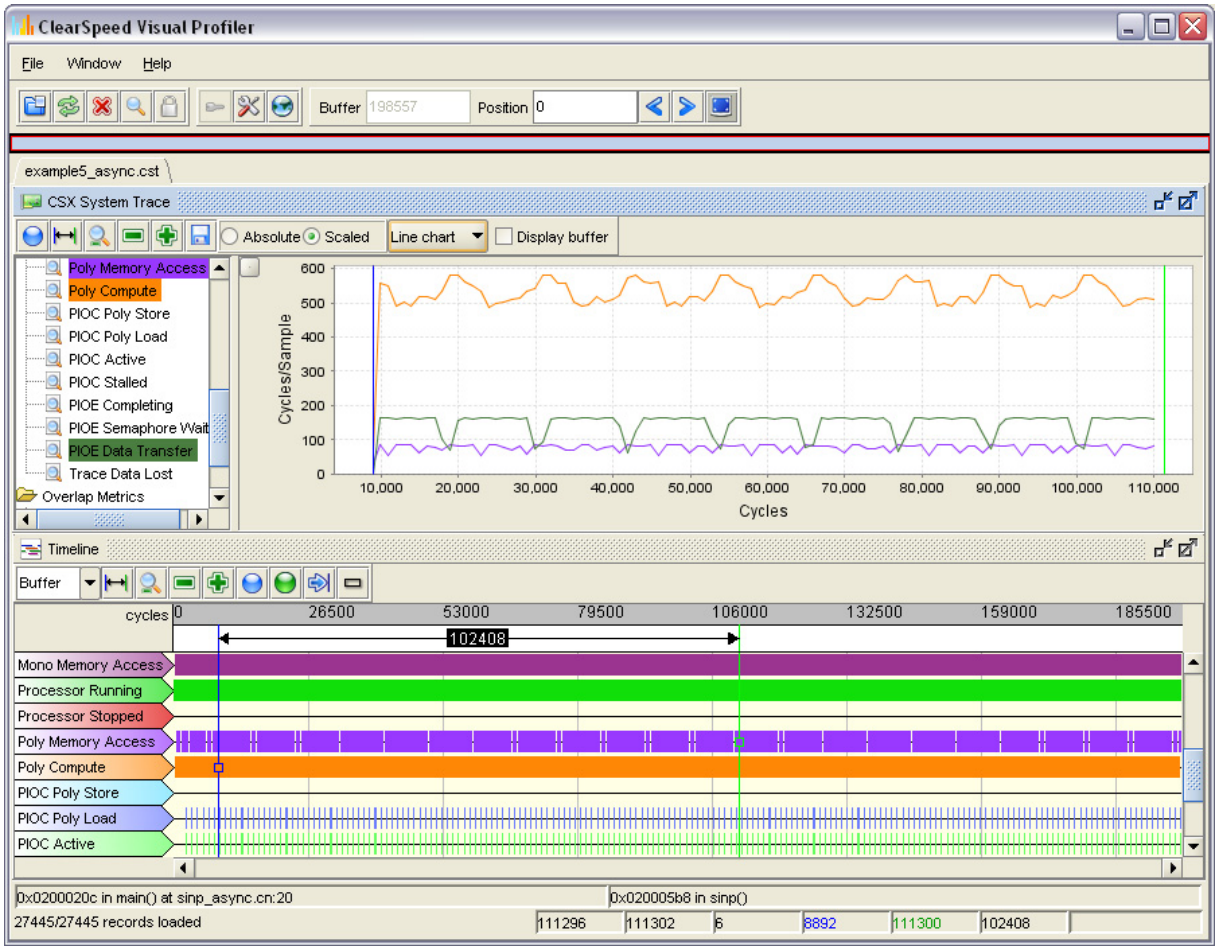

Figure 3.6: The ClearSpeed Visual Profiler enables an event log visualization for activity on the ClearSpeed processors. It shows in the upper graph how long one sample—in this case one instruction from the selected group—took to be processed. In the lower graph the utilization of the various processor function units is shown. [Cle10]

3.4.3 Convey

The Convey FPGA accelerator environment includes a Personality Development Kit (PDK) which allows to build fixed function blocks for the Convey environment [Con09b]. Since FPGA based performance analysis is difficult as it would require hardware modifications for the performance monitoring, the PDK contains a simulator so that current designs can be tested for correctness as well as performance.

3.4.4 NVIDIA Visual Profiler

The NVIDIA Visual Profiler is a platform independent performance tool for CUDA and OpenCL applications using NVIDIA GPUs [NVI11a]. It uses both event recording and sampling to generate performance profiles and timeline displays for GPU usage by using NVIDIA's public profiling tools API "CUPTI". The tool can also record and display host activity but is limited to working on one host. GPU applications are started from the GUI of the profiler and run on the same host. If a lot of performance data—typically hardware performance counter data—is to be recorded, the GPU application is rerun multiple times. In addition the tool tries to find typical performance problems and suggests optimization techniques.

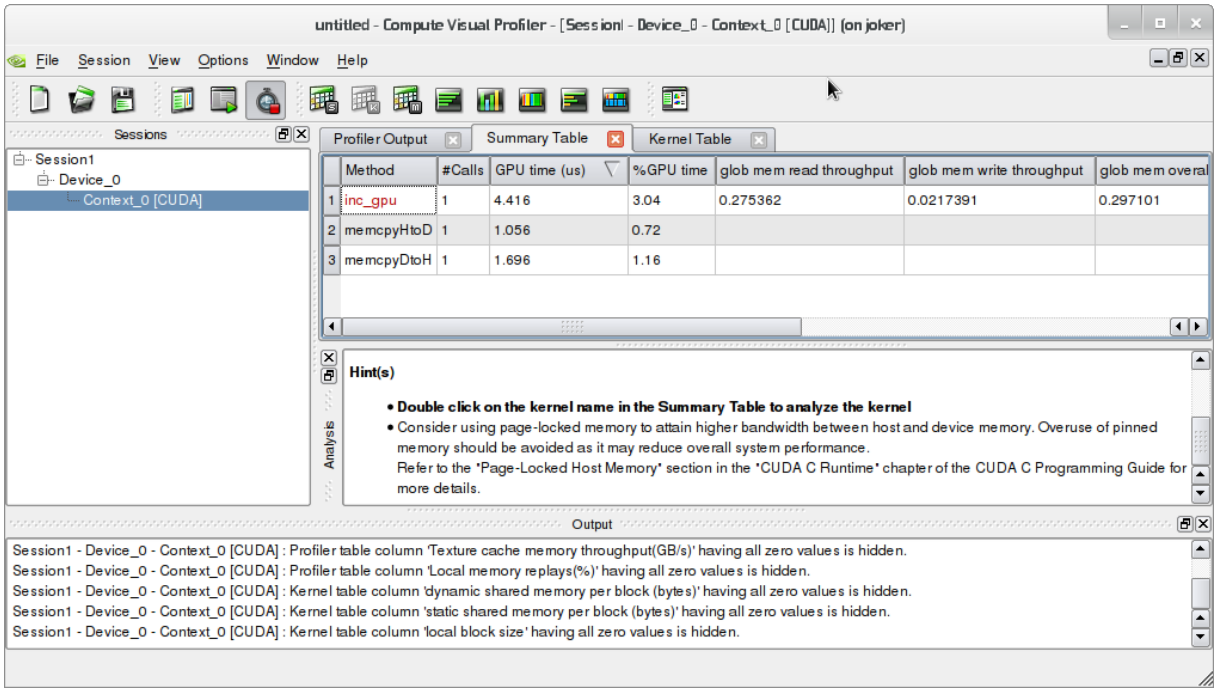

Figure 3.7: The NVIDIA Visual Profiler can present statistic profiles of kernels executed on NVIDIA GPUs. In this case it shows one of the CUDA SDK examples, where the kernel takes the longest time, followed by the two host-to-device transfers.

Among performance profiles the tool also provides a basic timeline view for the host-device interaction. Kernel statistics can be displayed after selecting a kernel in the table view of the tool. The granularity of the performance information is on a kernel level, even though performance counter data can be sampled at a higher frequency. The start and end times for kernels are recorded using special events within the execution streams for the kernels. A sample output of the profiler is shown in Figure 3.7. The profiler also has a command line equivalent to support profiling over slow networks or with high network latencies.

3.4.5 NVIDIA Parallel Nsight

Parallel Nsight is an extension to Microsoft's Visual Studio IDE [NVI11d]. It offers debugging as well as performance analysis functionality for interaction with NVIDIA GPUs. In contrast to the Visual Profiler it does not use the CUPTI API but rather the more powerful internal tools API. The tool supports performance analysis for CUDA, OpenCL, OpenGL, and DirectX. Parallel Nsight supports performance profiling of GPU kernels by sampling hardware performance counters, as well as by code injection to acquire memory usage statistics or floating point throughput [Hö12].

The presentation of the performance data is integrated into the Visual Studio GUI and offers timeline displays for the host-device interaction as well as a number of tabular or chart profiles. An example is shown in Figure 3.8.

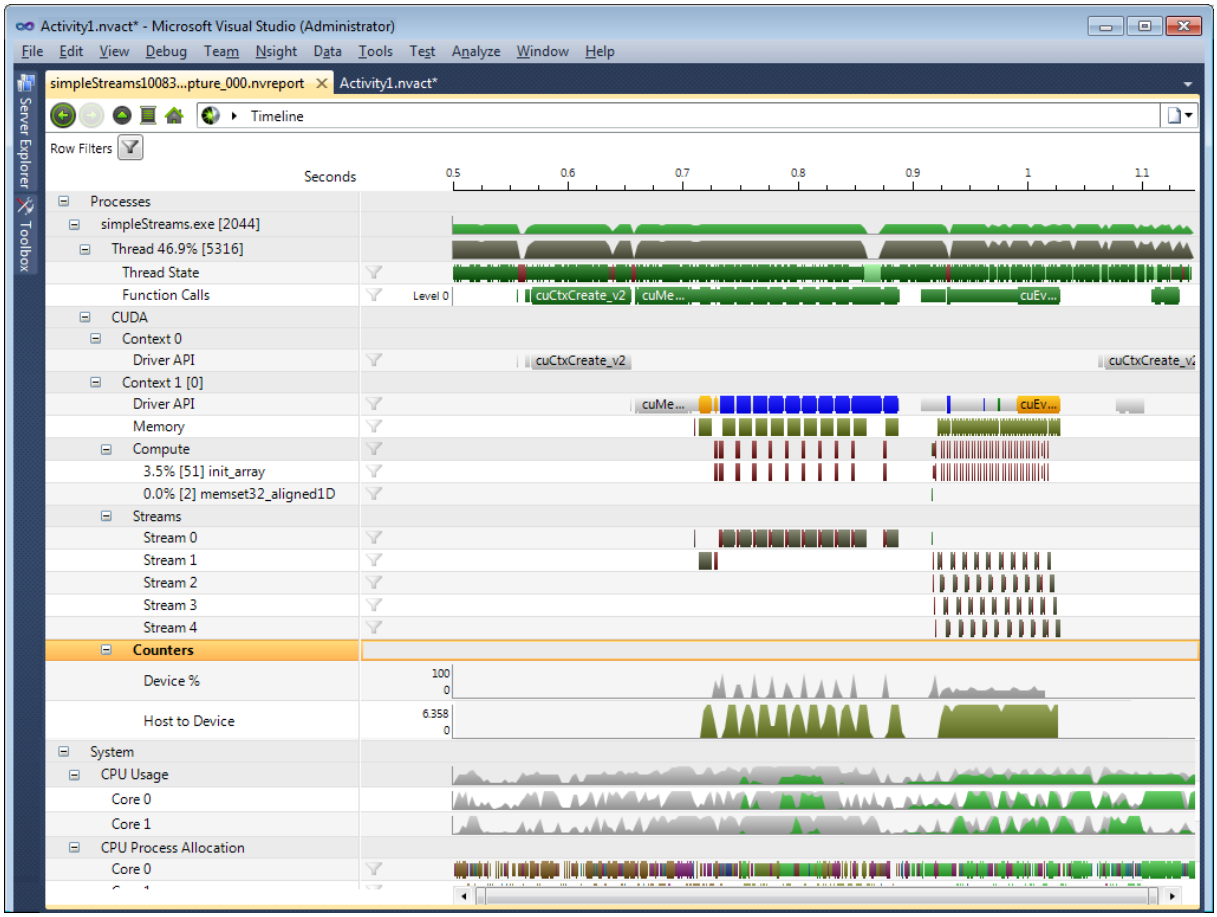

Figure 3.8: NVIDIA Parallel NSight enables a very detailed view into the execution of CUDA accelerated applications. It monitors both host and device activity and can display various performance aspects of both. In this case the CUDA activity in the two established device contexts as well as the compute activity in the GPU streams. [NVI11d]

3.4.6 AMD APP Profiler

The APP Profiler from AMD is a Visual Studio Plugin as well as a command line tool [AMD11]. It is designed to aid OpenCL and *DirectCompute* developers using AMD devices with performance analysis tasks. The tool can measure the execution time for device kernels and collect hardware performance counter data for AMD Radeon GPUs. The APP Profiler can visualize the recorded information in timeline displays and tabular profile information. It can show assembly instructions executed on the device and calculate device occupancy. An example of the tool output is shown in Figure 3.9.

3.5 Current Third Party Performance Tools with Accelerator Support

Aside from the performance analysis tools from the various hardware vendors that usually only support the vendor's hardware accelerators, third party performance tools typically offer a broader system view. Most research tools are published under an open source license as opposed to the closed source policy of

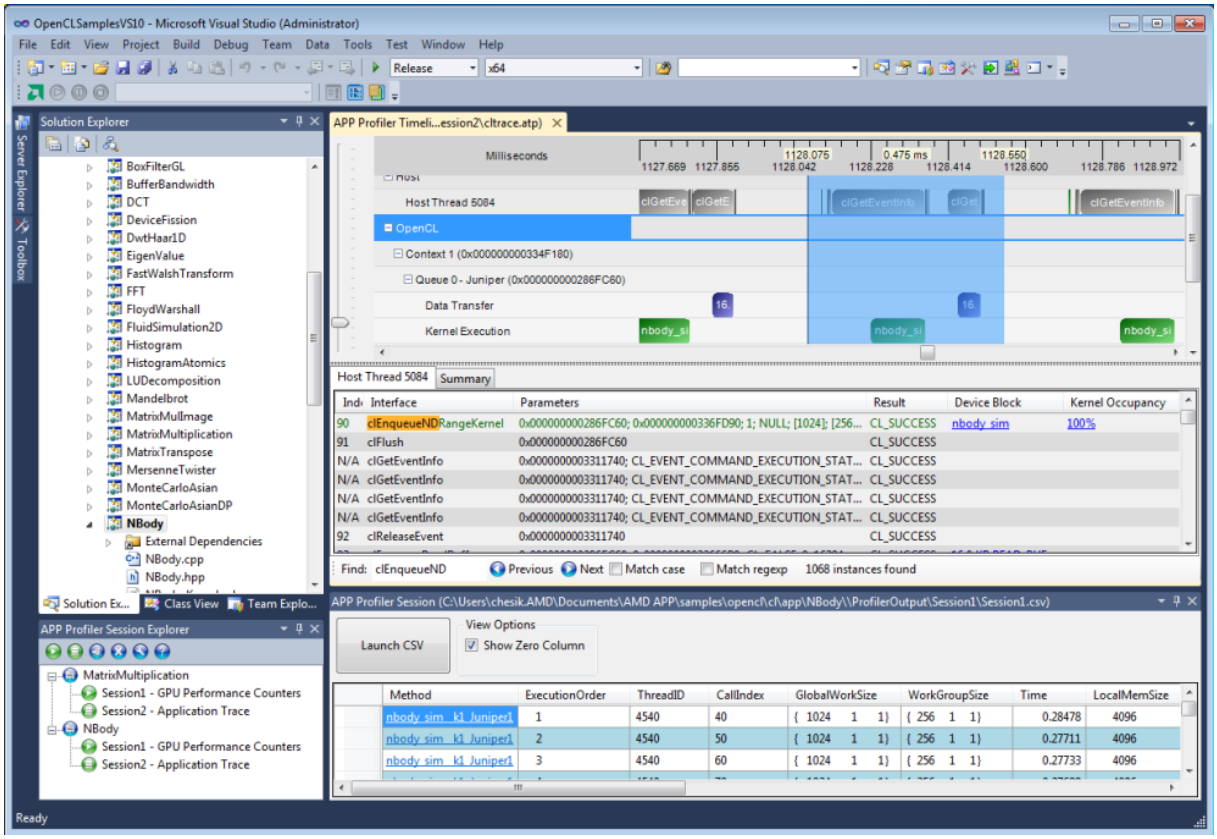

Figure 3.9: The AMD APP Profiler is a Microsoft Visual Studio plugin that enables performance analysis of OpenCL accelerated applications running on AMD CPUs or GPUs. The event log data can be shown as a timeline of various device states (top right), as a number of statistical information about the used OpenCL calls (middle right), or as kernel execution statistics (bottom right). [AMD11]

the vendor tools. While most such performance tools have their origin in the field of high performance computing and, thus, are tools that are able to record performance information for many concurrent processes, threads, and now also hardware accelerators, one exception for a pure GPU performance tool is introduced as well.

3.5.1 VampirTrace/Vampir

The Vampir performance tools are twofold: the event recording tool “VampirTrace” and the trace visualization tool “Vampir” [MKJ+08]. The two tools are joined by the used trace file format “OTF” [KBB+06] and also amended by the highly scalable analysis backend “VampirServer”. VampirTrace and the OTF-library are open source tools while Vampir and VampirServer are offered using a proprietary license. Since the VampirTrace/Vampir development is taking place at ZIH in Dresden as well, it was chosen as one starting point for implementing additions from this thesis, hence, it is introduced in more detail. A number of the developed additions have also been integrated into the final products.

Vampir is an acronym and stands for “Visualization and Analysis of MPI Resources” [NAW+96]. Its development started at the Research Center Jülich and moved to Dresden in 1997. There it has been augmented by the trace collector VampirTrace and the analysis backend VampirServer but also extended

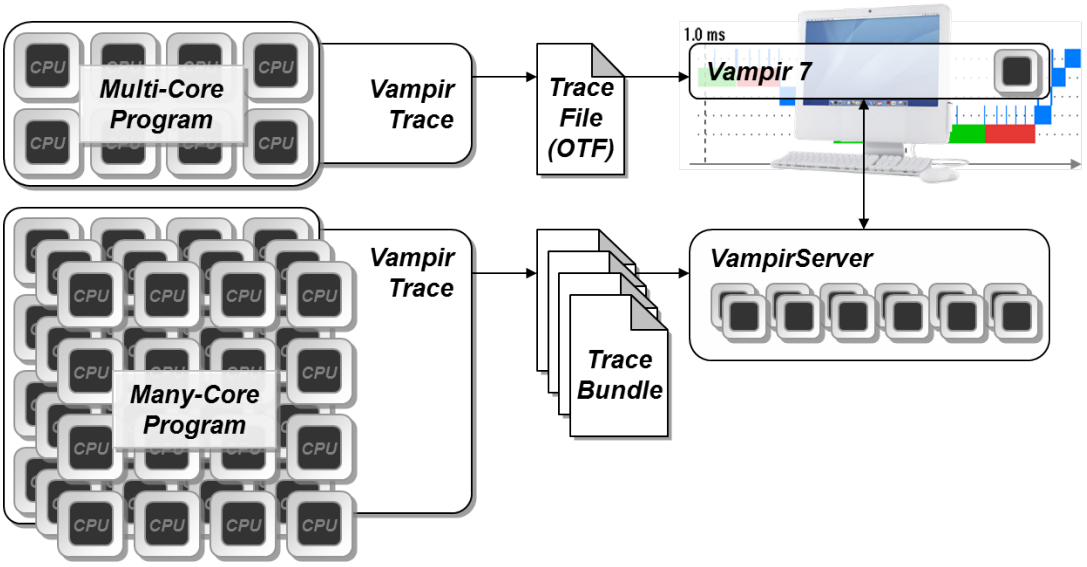

Figure 3.10: The workflow when using the Vampir tools is separated into an event logging phase using VampirTrace and an data analysis phase using Vampir. VampirTrace can record event log information on a very large scale. In the analysis of very large event logs, the analysis engine has to be detached from the GUI to mine the large amount of data fast enough for interactive analysis. [BHJR10]

to cover thread based parallism using OpenMP or pthreads. The analysis workflow using the tools is shown in Figure 3.10.

VampirTrace

VampirTrace is a program that allows applications to record performance events. The data can be recorded for sequential and parallel applications using MPI or threads which is one of the strengths of VampirTrace since it enables a very detailed study of all parts of a parallel application.

The event data is acquired by instrumenting the studied application. This can be done in three ways: manually, compiler based, or directly into the program binary. Manual instrumentation requires the programmer to add statements into his application's source code to call the VampirTrace API to record an event. This allows to trace any part of the application, e.g., individual loops but also large sections like program modules, thus, providing an arbitrary level of detail. It can also be combined with the other two instrumentation methods for additional detail. Compiler based instrumentation results in the compiler adding an entry and leave event at the begin and end of every function. For highly modularized C++ codes this leads to a lot of events being generated, thus, a large overhead. Dynamic runtime instrumentation uses the tool Dyninst [HB00] to insert calls to the VampirTrace library so that it records events of compiled applications. Due to highly optimized applications where compilers use techniques such as code inlining, this method of event generation complicates matching the performance events to the original source code.

When an instrumented application is executed, the inserted probes will trigger the collection of a time stamp which is stored together with the event data (type of event, name, and possibly additional performance data such as performance counter values) in an internal buffer. The buffer is flushed into the OTF trace file at the end of the application run or whenever it is filled to a certain threshold.

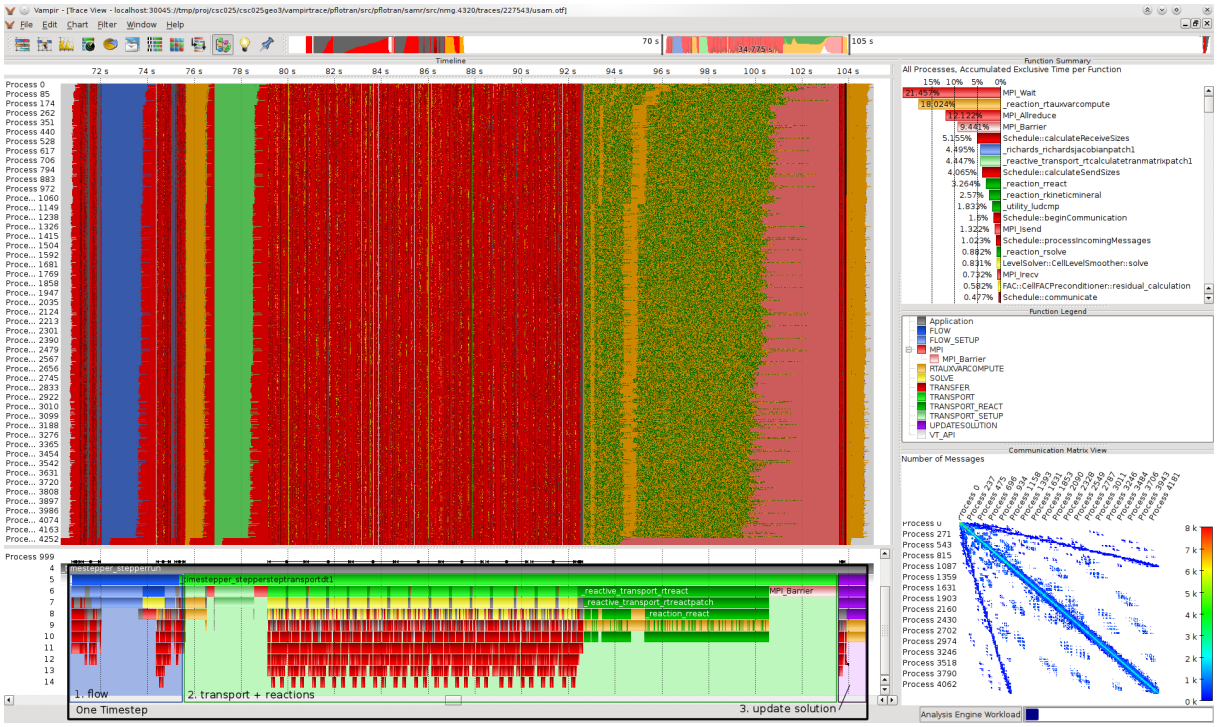

Figure 3.11: The Vampir GUI can present a lot of performance data at once. At the very top of the window is a thumbnail of the event log to enable quick navigation. On the right hand side are (from top to bottom) a profile on the execution time divided by function, a function legend explaining the used colors, and a communication matrix as already shown in Figure 3.4. On the left side is a timeline representation of all processes and below a timeline display for just one process that also shows the call stack levels of the selected process. [IHB11]

Score-P

Score-P is a joint project of Jülich Supercomputing Centre and the German Research School for Simulation Sciences, Technische Universität Dresden, Technische Universität München, University of Oregon, RWTH Aachen University, GNS Gesellschaft für numerische Simulation mbH, and GWT-TUD GmbH to provide a common performance monitor for a number of popular research based HPC performance tools [KRaM$^+$11]. It uses OTF2—the successor of OTF—as the common file format for event logs. Furthermore, the Score-P infrastructure also allows online aggregation and stores the profile data in the CUBE4 file format—a successor of CUBE3, the previous Scalasca file format. The Score-P infrastructure currently supports the following back-end tools for data analysis: Periscope (München), Scalasca (Jülich and Aachen), Tau (Oregon), and Vampir (Dresden). The CUDA event logging capabilities using the NVIDIA CUPTI interface developed as part of this thesis have been ported to Score-P and will become available in a future release.

Vampir

Vampir is an interactive visualization tool for OTF trace files. It consists of two parts: the data analysis backend and the visualization frontend. When a trace file is opened the backend reads the trace records and convert them into an internal data structure [Knü08]. The GUI offers multiple ways—called charts—to browse the performance data as shown in Figure 3.11. A unique feature of Vampir is the capability

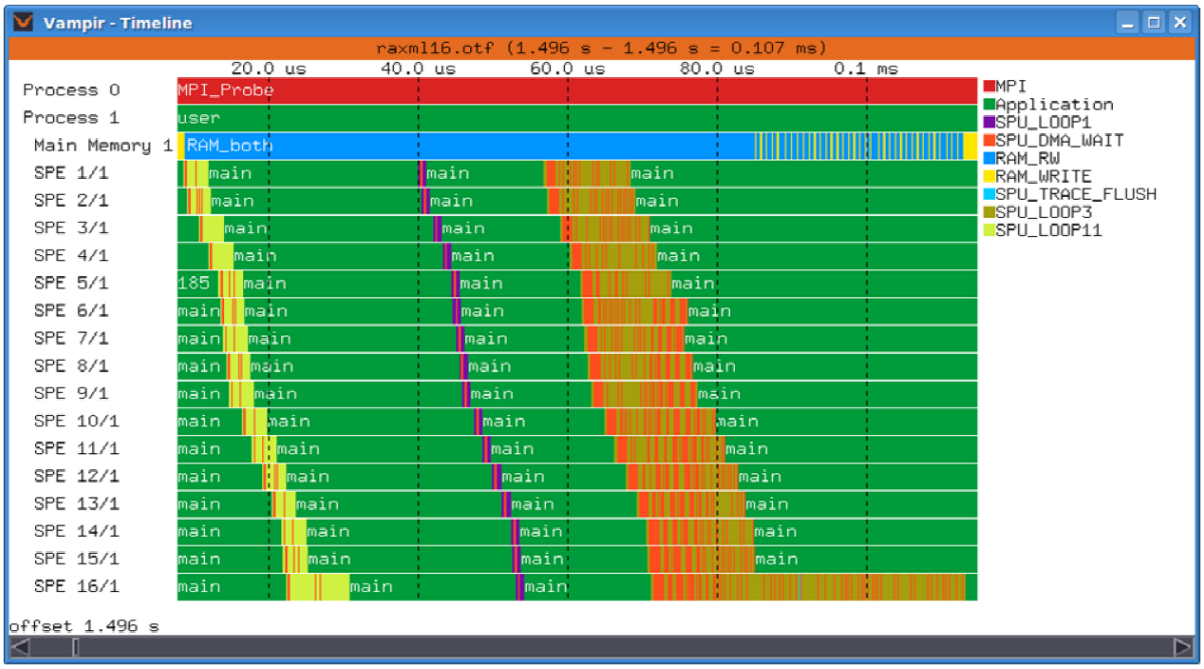

Figure 3.12: Vampir visualization of an event log of the RAxML application running on two Cell/B.E. processors [Hac08]

to offer a time based data view alongside various performance profiles. The user can navigate through the timeline displays by zooming and scrolling and the profile displays will always be updated to the currently selected time interval. In order to enable this interactive navigation all trace data has to be kept in main memory, thus, limiting the amount of performance data a single computer can handle. To overcome this limitation a highly parallel version of the analysis engine was developed and the GUI can also connect to such a detached backend [Bru07].

Special VampirTrace Solution for Cell/B.E.

HACKENBERG developed in [Hac08] a specific extension of VampirTrace which uses the following principles:

- Represent the SPE as threads of the PPE. Since the PPE launches the work of the SPEs, the behavior is similar to a CPU process spawning threads to offload work, thus, the analogy fits quite well.
- Represent mailbox messages and DMA transfers between the SPEs as well as the PPE as messages using a special communicator. Since the mailbox messages or DMA transfers transmits data from one memory location over a network (the EIB) to another location, this is also a valid analogy.
- Introduce the main memory as an artificial thread to provide a communication partner for DMA transfers to and from main memory.

The accelerator events are stored in the local memory of the SPEs and can, therefore, render an application using all of that memory unusable. The event buffers are transferred to the PPE which generates the full event log. When loading the event log with Vampir (as shown in Figure 3.12), the performance

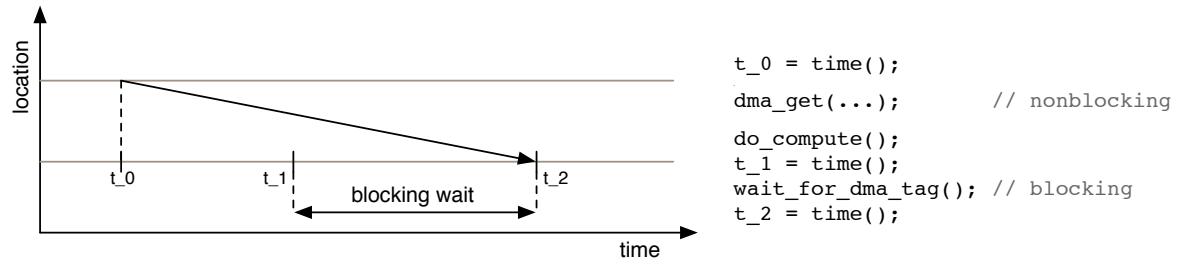

Figure 3.13: When logging DMA transfers on the Cell/B.E., three time stamps are taken. The three time stamps allow the determination of an upper bound for the actual transfer time. [Hac08]

analyst can see the hardware hierarchy of the Cell/B.E. processor and the recorded events with respect to their origin.

Additionally, a method is presented to derive an upper bound for DMA transfer durations in the Cell/B.E. enviroment. In this environment—as shown in Figure 3.13—an asynchronous DMA transfer is followed by a blocking wait operation that waits for the completion of the transfer. The three time stamps generated for the start of the transfer (t_0) as well as for the begin (t_1) and end (t_2)of the blocking wait operation allow the following conclusions about the actual duration of the transfer:

- If $t_1 \approx t_2$ then the transfer was completed before the wait operation, because the wait operation immediately returned. Hence, the maximum duration of the transfer is $t_1 - t_0$ or the time from the start of the transfer to the beginning of the wait operation. The minimum transfer time is > 0, since the transfer could have finished right after t_0.
- Otherwise the wait operation had to wait for the completion of the transfer ($t_2 > t_1$). In this case the maximum transfer time is $t_2 - t_0$ or the time from the start of the transfer to the end of the wait operation. The minimum transfer time is > 0 and smallest in the case where the transmission started right before t_2.

3.5.2 TAU

TAU—an acronym for Tuning and Analysis Utilities—is a very versatile performance tool [SM06]. It supports both sampling and event logging as well as a combined approach called event based sampling to record performance data [MMSH10]. An application is typically profiled by prepending the application invocation with the command `tau_exec` which launches the application under control of the TAU measurement system. It will record samples during the program execution and will generate a flat profile for the application. TAU supports performance analysis of MPI parallel, multi-threaded, and GPU-accelerated applications. For a more detailed analysis TAU also supports event logging. This method is also used for GPU event collection but requires callbacks from the accelerator library in order to record the device interaction [MBS+11]. Prior to the CUPTI API from NVIDIA (as discussed in Section 4.2.3 TAU used a proprietary kernel driver from NVIDIA to receive those callbacks [MBSM10, MMS10].

While TAU has a very powerful profile browser "ParaProf" built in (see Figure 3.14), it lacks a trace viewer. One possibility to view TAU traces is to convert them into the OTF trace format and explore

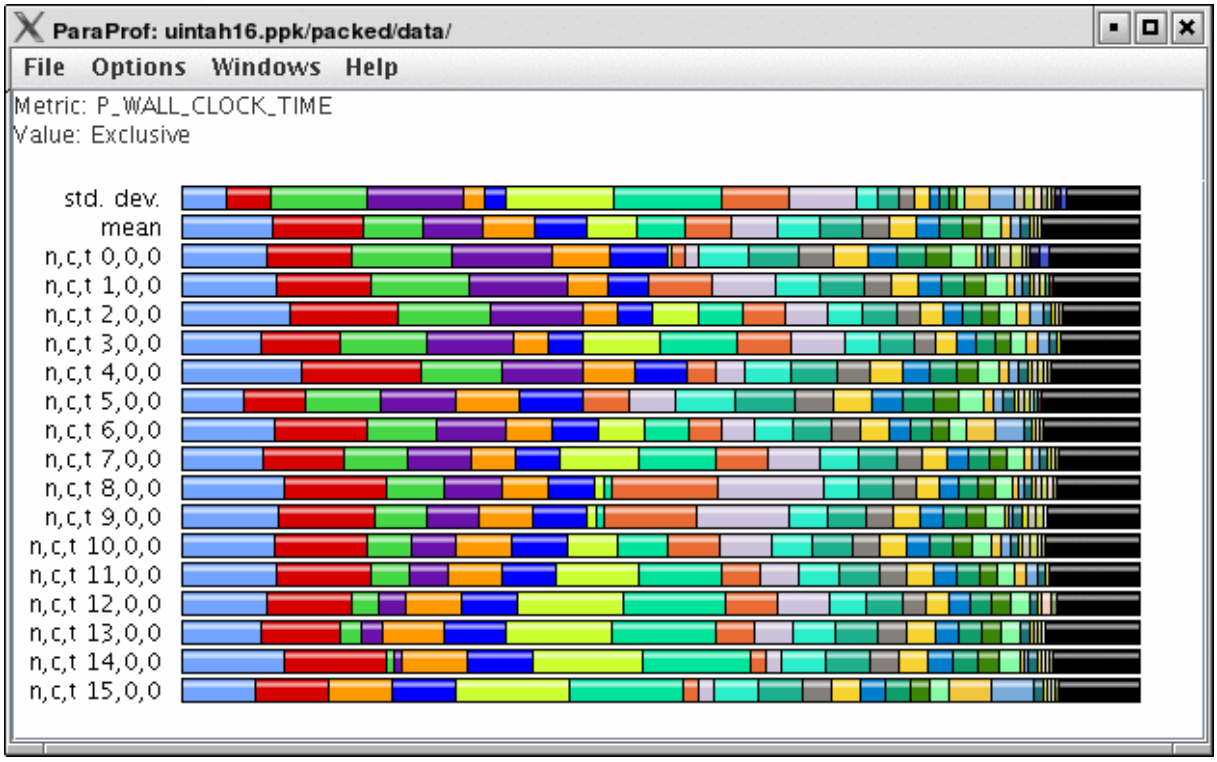

Figure 3.14: The ParaProf profiling display shows the function profile for all recorded processes. The mean values and standard deviations are shown on top of the display. In this case a very strong load imbalance between the recorded processes can be seen for most functions. [SM06]

them with Vampir. Another option is to convert them to the SLOG trace file format and use the tool Jumpshot [ZLGS99] to display the trace data. TAU also supports the library wrapping approach that has been developed as part of this thesis and which was published in [DIJ10].

3.5.3 Integrated Performance Monitoring (IPM)

IPM is an event based profiling tool [Ski05, FWS10, FWS11]. It collects performance data for MPI, multithreaded and CUDA accelerated applications by intercepting the calls to the corresponding system libraries. This is achieved by directing the dynamic library loader "ld" to preload the IPM-library. Every invocation of one of those libraries is caught by IPM and immediately added to a performance profile, thus, generating only a small overhead to the execution time of the application.

The performance profile is stored as an XML file which can be turned into an HTML report or likewise into a CUBE file to be opened with the CUBE GUI which is part of the Scalasca tool set [GWW+10]. The report includes flat profiles, histograms of process profiles, and various charts on the distribution of used library routines. As a result, the reports allow an easy way to compare multiple program runs as well as a first hint for potential hot spots in the code that should be studied in more detail.

3.5.4 CEPBA MPItrace

The European Center for Parallelism of Barcelona (CEPBA) develops a set of performance tools designed for high performance computing: Paraver (a trace analyzer), Dimemas (a simulator), and various tracing tools for different parallelization strategies [Lab10]. The CEPBA tools' only support for hardware accelerators comes for Cell/B.E. with a special version of MPItrace that records performance events for later visualization with Paraver [Cen07].

3.5.5 PAPI

A standard technique for performance analysis of CPU based applications is the usage of CPU hardware performance counters. They provide counts for a number of performance related events like a floating point operation count, number of cache hits and misses, as well as memory subsystem usage. A popular and platform independent way to gather this data is the Performance Application Programming Interface (PAPI) [TJYD10]. This interface distinguishes between high level and native performance counters. While the latter maps the counters that are available on the current platform into the interface and, thus, provides a performance monitor access to this device specific data, the former provides a platform independent high level event which the PAPI library itself maps to the correct native event. As a result a performance monitor can use the high level events to provide a platform independent way of accessing very detailed performance related hardware information. Starting with version 4 of the tool, it was modularized and called Component PAPI (PAPI-C). PAPI-C allows collecting measurement data from multiple sources simultaneously, thus, extending the library to collect more than CPU counter data. PAPI-C has been extended to also collect data from then sensors plugin to capture environmental data from the computing host as well as Infiniband counters. Most recently PAPI-C can read GPU performance counters from NVIDIA devices using the CUPTI API [MBS+11].

3.5.6 GPU Ocelot

GPU Ocelot is a GPU assembler modification framework for CUDA applications [KDY11]. It takes the compiled CUDA kernels in its PTX intermediate representation and allows multiple modifications to it. For one the tool can translate the PTX code to support other target architectures (like CPUs, AMD GPUs, or a PTX emulator). Furthermore it can instrument the PTX code to generate performance reports.

3.6 Classification of the Existing Performance Tools

The presented performance tools that support monitoring at least one kind of accelerator was already divided into two groups: vendor provided tools and third-party tools. Table 3.1 compares the presented tools with respect to their supported techniques and monitoring capabilities. It can be seen that VampirTrace currently offers the highest scalability among the various performance monitors.

Table 3.1: Overview of the presented performance monitoring tools

Performance Tool	License	MPI Support	Thread Support	Accelerator Support	Programming Models	Monitoring Method	Scalability
IBM Cell/B.E. PDT	proprietary	yes	yes	Cell/B.E.	Cell SDK	Event logging	1 host
ClearSpeed Visual Profiler	proprietary	no	no	CSX Processors	Clearspeed SDK	Event logging	1 host
Convey PDK	proprietary	no	no	Convey FPGAs	Convey PDK	Simulation	1 host
NVIDIA Visual Profiler	proprietary	no	yes	NVIDIA GPUs	CUDA, OpenCL	Event + sample logging	1 host
NVIDIA NSight	proprietary	no	yes	NVIDIA GPUs	CUDA, OpenCL	Event + sample logging	1 host
AMD APP Profiler	proprietary	no	yes	AMD CPUs+GPUs	OpenCL	Event + sample logging	1 host
VampirTrace	open-source	yes	yes	NVIDIA GPUs, any OpenCL device	CUDA, OpenCL	Event + sample logging	>200,000 event streams
Score-P	open-source	yes	no	NVIDIA GPUs (next release)	CUDA	Event + sample logging	≈8,000 event streams (file system limit)
TAU	open-source	yes	yes	NVIDIA GPUs, any OpenCL device	CUDA, OpenCL	Event + sample logging	≈100 event streams (event logging limitation)
IPM	open-source	yes	yes	NVIDIA GPUs	CUDA	Event aggregation	>10,000 hosts
CEPBA MPItrace	registration	yes	yes	IBM Cell/B.E.	Cell SDK	Event logging	≈1,000 hosts
PAPI	open-source	no	no	NVIDIA GPUs	CUDA	Sample aggregation	1 host
GPU Ocelot	open-source	no	no	NVIDA GPUs, any OpenCL device, x86 CPUs	CUDA	Event aggregation	1 host

4 A Generic Method for Extending Performance Analysis to Hardware Accelerators

The last two chapters provide the foundation for this thesis upon which a general approach for recording performance events from any kind of accelerator is set. A generic host-based performance API is derived and one example of its implementation is shown in cooperation with NVIDIA. In addition, it is also investigated how the performance analysis can be extended to also record data directly on the accelerator. The second part of this chapter discusses how the additionally recorded data about accelerator usage is added to the data presentation by the existing tools. After that a method to present event logs based on program states is introduced as a suggestion to overcome certain disadvantages of the existing presentation methods.

4.1 Automatically Intercepting Library Calls

Chapter 2 noted that all current accelerators are host driven in such a way that a host program interfaces with an API to offload work onto the accelerator. The focus of this section is how to derive a flexible method of intercepting this API using a performance monitor in order to exploit this commonalty.

4.1.1 Framework for Logging any API Invocation

The usage of a system library for any kind of program follows a simple principle: an API defines a programming interface and a semantic how the underlying system library will respond to input to the defined subroutines. The programming interface is typically a header file which is included by the compiler's preprocessor into the program that uses the API. After the compilation of the program the linker will connect the program with the system library. This can happen in two ways: via static or dynamic linking. The former will include the whole system library into the binary program and, thus, increases the size of the compiled executable program. The latter only checks whether the libraries are available as dynamically linkable objects and connects the executable when it is started with the necessary system libraries. The usage of any system library by the program-under-test can be monitored by injecting a monitoring library that contains the same symbols as the original system library in between the program and the library as shown in Figure 4.1. If the program-under-test calls a routine from the system library, it will enter the subroutine with the same name in the monitoring library. There, the necessary event logging can be done before passing the call to the original library.

One possible method for the monitor injection is the linking process at the end of the compilation. The `ld` linker of any GNU based operating system supports this with the `--wrap` flag. If the program-under-test tries to call `foo` in a system library and the linker is instructed to wrap `foo` via `ld --wrap foo`, the

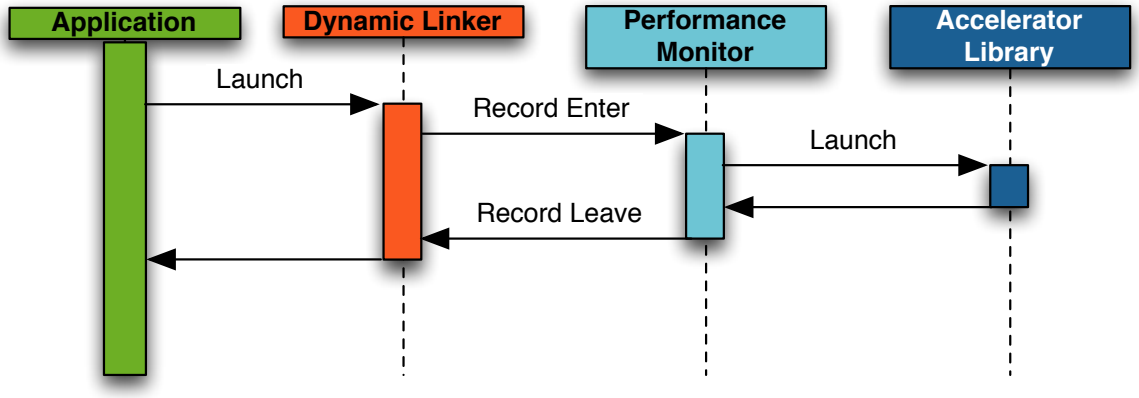

Figure 4.1: Library calls will be intercepted by a performance monitor, if the monitor provides an own library with the same function symbols as the library that is to be intercepted. The dynamic linker has to be instructed to also load the performance monitoring library. When the program-under-test is then started, the performance monitor is called by the application for each original library invocation, can record the library usage, and call the original library.

linker will direct the call to `foo` to the symbol `__wrap_foo`. This routine is provided by the monitoring library. It logs the desired events and invokes the original `foo` via the symbol `__real_foo`. The following example illustrates the usage:

```
#include<vt_user.h>  // event triggers

void * __wrap_foo () {
   void *ret;  // record return value of original "foo"

   VT_USER_START("foo"); // record begin of "foo"
   ret __real_foo();
   VT_USER_END("foo"); // record end of "foo"

   return ret; // return return value of original "foo"
}
```

MUELLER ET.AL. used this static method to aid the analysis of I/O behavior of applications [VMMR09]. The principle, however, is much more powerful. If it is combined with a code parser and an automated code generator, it will enable the monitoring of any arbitrary API usage. The necessary steps are explained in the following.

Since the program-under-test uses a header file as part of the API to a system library, the monitor can use the same header file to generate its wrapper library. A C parser produces a list of all subroutines that are available via the header file. Some libraries contain a lot of subroutines—the Intel Math Kernel Library (MKL) provides well over 1,000 routines via the `mkl.h` header file. Hence, a filtering mechanism that selects or deselects certain subroutines is highly useful. The list of selected subroutines is then passed to the code generator to produce the wrapper library. For every subroutine in the list passed from the first step the code generator will create wrapper routines as shown above for the subroutine `foo`. Furthermore, the list of substituted subroutines needs to be supplied to the linker so that it can wrap them. The generated wrapper library will trigger events for every entering and leaving of the system library.

Dynamic Linking

The injection approach using the wrapping functionality of the linker has the shortcoming that it requires recompiling the program-under-test. If the program uses dynamic linking, the wrapping technique can be extended to make recompilation unnecessary. In this case the wrapper library can use the same symbol names as the targeted system library and the `LD_PRELOAD` functionality of the linker for symbol substitution. The wrapper library then uses a library (`dlfcn.so`) provided by the linker to handle the invocation of the original library. The `dlfcn` library provides the routine `dlsym` to extract symbols from shared libraries and, thus, make the routines from the library available. In this way, the program monitor can substitute symbols dynamically but preserves the correctness of the overall program execution. The same method has for example been used by MICKLER in a special solution for I/O analysis [Mic07]. The following code example illustrates the principle:

```
#include<vt_user.h>  // event triggers
#include<dlfcn.h>  // dynamic library handling

void *libhandle = NULL;  // pointer to hold the address of the original library

void (*real_foo) (void) = NULL  // function pointer for real "foo"


void __attribute__((constructor)) lib_init() {
   // will be called upon loading the library at the start of the program

   linhandle = dlopen("libfoo.so", RTLD_NOW);  // now open the real library containing "foo"

   *(void **)(&real_foo) = dlsym(libhandle,"foo"); // have real_foo point to the original foo

}
void * foo () {
   void *ret;  // record return value of original "foo"

   VT_USER_START("foo"); // record begin of "foo"
   ret real_foo(); // invoke original "foo" and store return value in "ret"
   VT_USER_END("foo"); // record end of "foo"

   return ret; // return return value of original "foo"
}
```

This straightforward approach combines the precision of events with the simplicity of sample based performance evaluation methods since it can be used on unmodified programs. The IPM profiling tool is purely based on this approach and substitutes a number of common HPC libraries to collect statistics about their usages [FWS10]. However, the approach presented in this thesis is much more powerful since it is not limited to previously known library routines. It rather offers an automated workflow for intercepting arbitrary library calls to exactly determine how much time the program-under-test spends using functions provided by an external library. The workflow was proven in [Ils09] and has been included into the VampirTrace environment with version 5.8.

Example: Understanding Math Library Usage

The **bicgstab** method for iteratively solving a system of linear equations is a good example for the ease of the presented performance analysis method for arbitrary library usage. The bicgstab method as presented in [Saa03] is especially suitable for sparse linear systems and typically utilizes a mathematical library for the sparse matrix-vector-operations needed. If the performance analyst is interested in how well an

implementation of the algorithm actually performs, the time spent in the library routines is not recorded in case a classical event triggered measurement approach is used. Thus, it is necessary to intercept all calls to the mathematical library. Furthermore, it would be possible to start an event triggered measurement without instrumenting the program-under-test in a first experiment.

In the studied example the code utilizes Intel's Math Kernel Library (MKL) which can be determined by running the program `ldd` against the program-under-test to reveal its dependencies to shared system libraries [Ils09]. In a first step the analyst runs the wrapper library generation against the `mkl.h` header file of the MKL. Next, the generated wrapper library is injected into the dynamic linking process by placing it into the `LD_PRELOAD` environment variable and the program is executed. This links the wrapper library as well as the performance monitor against the application and generates an event log of how the program-under-test interfaces with the MKL. It does, however, not reveal any internal information about the program-under-test itself. The workflow is briefly shown in the following console log:

```
juckel@joker:~> vtlibwrapgen -g MKL -o mkl_wrap.c \
   /sw/global/compilers/intel/12.1/mkl/include/mkl_spblas.h
Generating library wrapper source file
 Parsing input header file
 Processing function declaration statements
  100.00 %  done
 Created mkl_wrap.c
Done
juckel@joker:~> vtlibwrapgen --build --shared -o libmkl_wrap mkl_wrap.c
Building wrapper library
 Compiling library wrapper source file mkl_wrap.c
 Linking libtool object file mkl_wrap.lo
 Created libmkl_wrap.so
Done
juckel@joker:~> LD_PRELOAD="./libmkl_wrap.so" ./a.out
...
```

The timeline and execution profile of the generated event log can be seen in Figure 4.2. It is clearly visible that the application heavily utilizes the MKL and is not spending much time in its own code. If one neglects the initialization and finalization of the program, 86 % of the execution time is spent in the mathematical library. Hence, the focus for performance optimization for this program should be in locating the most capable mathematical library for the utilized sparse matrix-vector-operations.

The generated workflow for intercepting any kind of libraries proves to be highly valuable in easily determining the amount of time a program actually spends running its own code instead of using system libraries.

4.1.2 Additional Semantics to Automatically Generated Library Wrappers

The automated generation of wrapper libraries is a useful foundation to extend this approach to hardware accelerators. One conclusion of Chapter 2 is that all accelerators are currently accessible via a host-driven programming API. A commonality among all those APIs is that at some level they will involve a system library to communicate with the accelerator. The interaction of the program-under-test with this library can be monitored in the same manner as introduced in the previous section. Figure 4.3 shows the result for a simple CUDA accelerated program monitored by using this approach. It illustrates another feature of accelerator APIs which in turn makes performance analysis more difficult: the function calls in the library are typically asynchronous. As a result the time spent by the program-under-test in the

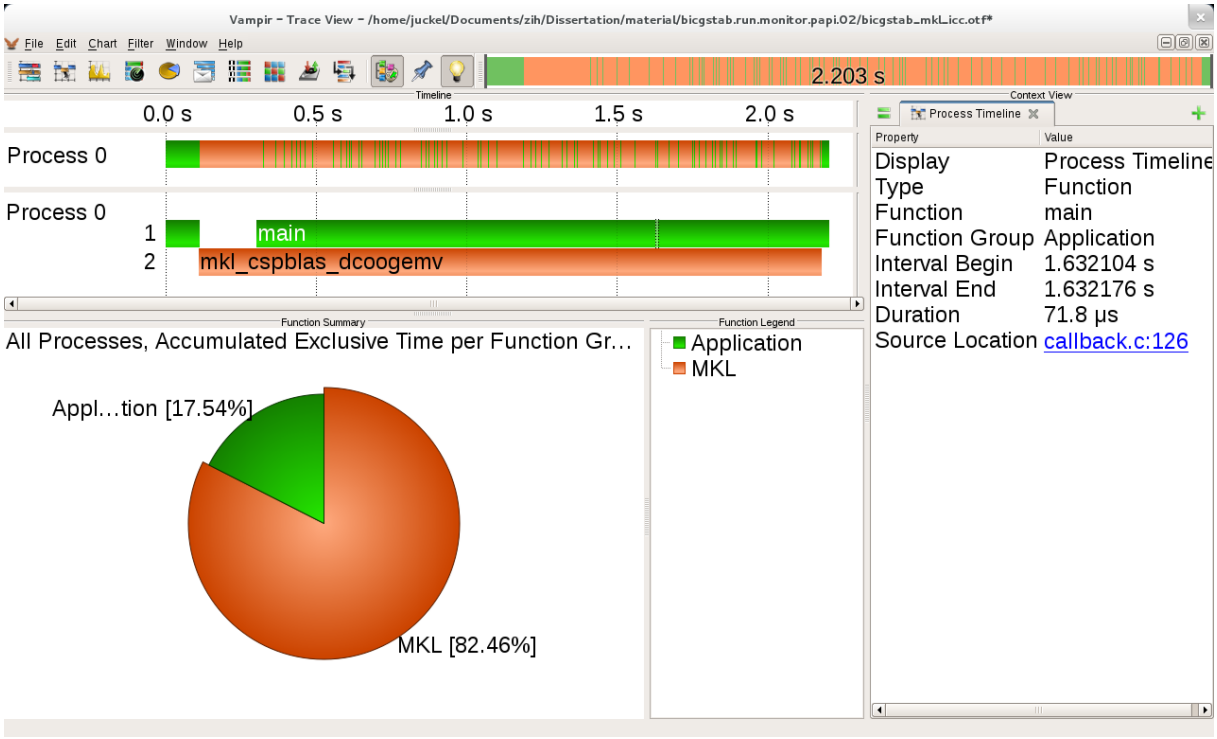

Figure 4.2: The analysis of the bicgstab application reveals that most of the overall execution time (82.46%) is spend within the MKL.

accelerator API does not easily reflect the time the accelerator is utilized. A more complex interpretation of the triggered events is required to provide insight into how well the accelerator is used.

Fortunately, the available event triggers are enough to derive an upper bound for the accelerator activity durations. Furthermore, existing established performance data representation techniques can be extended to hardware accelerators as well. Using the solution developed by HACKENBERG presented in Section 3.5.1 and the similarities between the different accelerators as shown in Section 2.4 as a basis the following principles can be established:

- Represent the execution stream launched by the host on the accelerator as a thread launched by the host process. Doing so preserves the parent-child-relationship between the host and the accelerator and enables a logical timeline based representation in the later analysis.
- Represent data transfers between host and device memory as messages using a special communicator. Since a (MPI) message is in fact a transmission of data from one physical memory location over some kind of network to another memory location, this generalization could in fact be seen as a message as well.

The main difference between the specific Cell/B.E. approach and this more general representation is the nature of the accelerator event streams. While for the Cell/B.E. they represented threads running on the physical processors (SPEs), they in general represent logical execution streams on the accelerator. This allows for situations where one execution stream is encompassing the whole accelerator device, which would just be represented as one thread. At the same time it enables concurrency on the device by allowing multiple execution streams to share the device, as it is possible in reality as well.

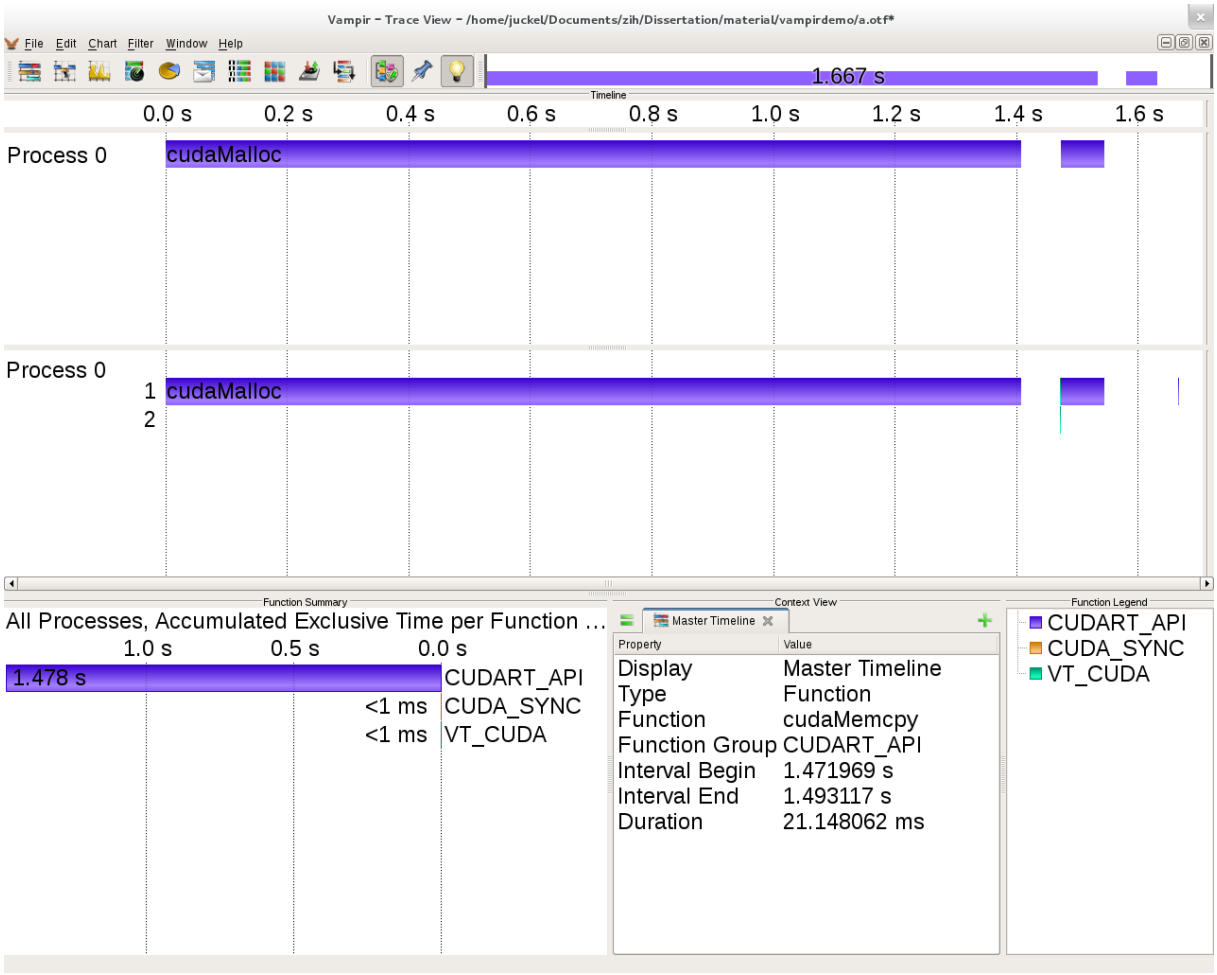

Figure 4.3: When using the automatically generated CUDA wrapper, the execution of a simple CUDA accelerated application containing just one kernel will generate an event log of the CUDA runtime API usage. This event log is dominated by the long device initialization and the extremely short asynchronous call for a kernel launch does not provide any information about the actual execution time of the kernel.

Reusing these already established metrics enables an easy integration into existing tools that already have means to record thread usage and message activity. The term “metric” is used in this thesis to generalize a recorded measurement category, such as messages. The flipside of the “reusage” coin, however, is that the presentation tools could previously safely assume a metric to be unique. If the tool extension now overloads an existing metric with additional data, the downstream profile generation of the performance data will mix the different data for the same metric together and, thus, disturb the analysis process. A filtering technique is needed to select only certain threads or messages for analysis to still provide valuable performance profiles. Most interestingly, a timeline representation of the event log still provides a logically correct presentation of the performance data even with the overloaded metrics. It basically adds another dimension to the data since processes of the program-under-test can now communicate with one another but also with their respective accelerators.

Considering the asynchronous nature of hardware accelerator communication makes it difficult to correctly determine when exactly certain events occurred on the accelerator by only monitoring the host activity. The approach introduced in Section 3.5.1 can serve as a blueprint for handling the asynchronous

nature of the hardware accelerator interaction. In order to ensure that all data is available in the device memory prior to launching an accelerator program, a synchronization operation has to be inserted before. The synchronization can be achieved by an explicit synchronization call or a synchronous data transfer. The computation has then in turn to be completed before the results can be returned to the host which can be ensured by a specific wait operation for the host. Both possibilities also highlight that asynchronous activity on the device can occur on two independent levels: data transfers and computation. Yet, synchronization operations are used frequently to ensure correctness of the computation and, thus, provide an anchor for deriving the maximum durations. It is certain that every similar activity (computation or transfers) has been finished at that point. Since the accelerator APIs allow multiple asynchronous operations up until this fixed synchronization point, deriving the potential execution times is very complicated and calls for a more sophisticated API assisted method of time stamp inquisition.

4.1.3 CUDA/OpenCL: Exposing Details on the API Level

The CUDA runtime API is a high level programming method to run application parts on graphics processing units. It has all the general characteristics that have been introduced previously: it is a host-driven, asynchronous programming interface that uses dedicated data transfers between host and device memory. OpenCL is quite similar to the low level method for running application parts on graphics processors by using the CUDA driver API. It does, however, target a wider range of accelerators and currently supports Intel and AMD CPUs, AMD and NVIDIA GPUs, Altera FPGAs, and IBM Cell/B.E. processors. Both OpenCL and the CUDA driver API are using the C99 standard of the C programming language and require no further extensions. The CUDA driver API is a superset of the runtime API which in turn abstracts the central routines of the driver API into programming language extensions. A commonality between all three approaches is the usage of a system library to communicate with the accelerator driver. Hence, the workflow that has been described in this section can be applied to all three as well.

As a first step the library wrapper generator presented in Section 4.1.1 was used on the `cuda_runtime.h` and `opencl.h` header files to generate the first versions of the performance monitors for both acceleration methods. Using the techniques established in Section 4.1.2, the device activity can be derived by monitoring the host's API usage. The first monitored API call creates an additional thread in the event log for the host process to hold the performance events generated for the device. This artificial thread in the event log records the execution time for accelerated program segments and serves as a source or target for data transfers. One difficulty with CUDA and OpenCL accelerated applications is the concept of parallel execution on the accelerator device. This is realized by multiple execution queues—in CUDA called streams—into which the various kernels and memory transfer operations are enqueued. For each queue, however, only one kernel can be running at every point in time. Hence, the concept of one artificial thread can be extented to store the performance events for one stream.

Another requirement for an API-based performance monitor for CUDA and OpenCL is a precise timing of the events on the accelerator such as kernel execution or asynchronous data transfers. Fortunately both APIs offer an event mechanism to synchronize multiple execution queues such that the start of a kernel depends on the completion of another. These events allow the generation of a global device time stamp. Hence, an asynchronous accelerator activity like a kernel execution can be surrounded by timing events

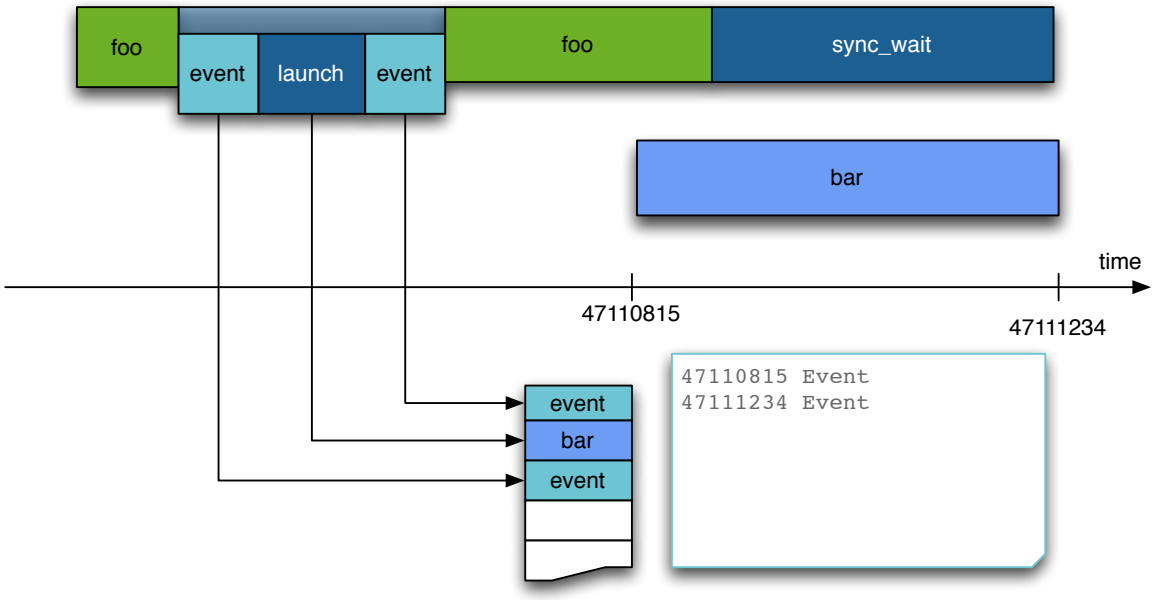

Figure 4.4: The extended launch function in the CUDA runtime / OpenCL wrapper library can be used to insert timing events before and after the actual kernel launch. This will generate time stamps directly on the device which can be read and translated into host time stamps during a synchronous operation by the host.

such that right before and after the kernel execution a device time stamp is collected as shown in Figure 4.4. The synchronization of host and device time is done by also bracketing synchronous API calls with those timing events. For a synchronous function call all four time stamps are known: host and device start time as well as host and device end time. Thus, the device time can be easily converted into host time. Doing this throughout the program execution ensures the compensation for clock drift between the two separate clocks. The device time stamps can then be used to populate the artificial thread with correctly timed device events. The result is a complete picture on a kernel level how a CUDA/OpenCL application is utilizing the accelerator device as shown in Figure 4.5.

The overhead for using library wrapping and to inject events into the execution streams for CUDA is shown in Table 4.1. The flush operation which converts prerecorded parameters of kernel launches and data transfers is only called once per 1000 iterations and has a constant overhead. Since the execution times vary quite a bit, the best approach to determine an overhead is to use the average. These numbers already expose a trend: kernel invocations induce a smaller overhead than asynchronous memory transfers and monitoring on the newer Fermi GPUs (S2050) has a lower overhead than on the older Teslas (S1070).

4.2 Design of a Host Based Accelerator Performance API

The previously introduced pure API interception approach has the advantage of being very flexible and can, thus, be easily adapted to different accelerator environments. The only dependency is the need for a library to intercept calls and a device based timing routine to provide time stamps for accelerator

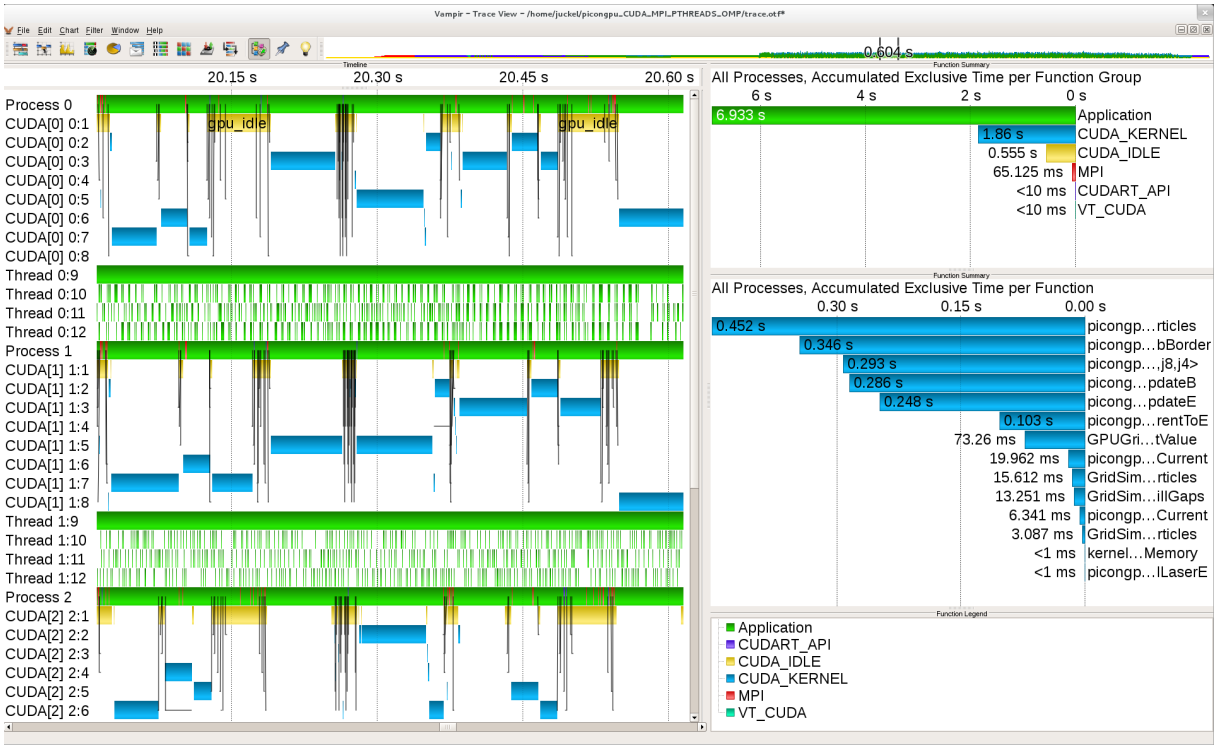

Figure 4.5: The Vampir view of CUDA activity of a multi-hybrid application using API based CUDA tracing shows how the host processes interact with the multiple streams on the devices. The profile shows the most time consuming CUDA kernels in the current zoom window. The timeline shows all levels of parallelism: three processes, each using eight CUDA streams and four host threads.

Table 4.1: The Monitoring Overhead for the CUDA Runtime API using the presented library wrapping approach is measured measuring 100 times 1000 repetitions of each event using the CUDA Toolkit version 4.2 and the CUDA Driver version 295.41.

GPU Type	Event Type	Execution Time $[ms]$			Overhead per Event $[\mu s]$			Flush $[\mu s]$
		min	avg	max	min	avg	max	
S1070	Kernel	16.748	16.818	16.827	11.283	**11.350**	11.344	1409.714
S1070	Memcpy	138.615	150.070	175.128	2.685	**21.921**	22.649	2343.981
S2050	Kernel	9.081	9.258	18.448	3.810	**6.466**	6.666	1655.198
S2050	Memcpy	20.390	39.593	40.403	20.208	**20.535**	22.590	2845.277

based events. This approach, however, has some flaws: the main problem is the device timer. On NVIDIA GPUs the device timing event is realized as a global device event and serializes all parallel stream execution. This results in an severe perturbation for applications utilizing parallel execution on the accelerator device. Furthermore, the accelerator device has additional information about the execution of an accelerated program which cannot be collected by just intercepting library calls. This section discusses the design of a host based event logging API for detailed accelerator analysis and presents how this design was realized as part of the NVIDIA CUDA Profiling Tool Interface (CUPTI).

4.2.1 Additional Device Performance Data

Aside from the monitored data transfers and kernels, the accelerator itself offers additional performance metrics that are of interest to the performance analyst. Most importantly, it typically offers a better timing routine than the previously used global timing event with its side effect of serializing parallel kernel executions. Since the accelerator device usually has its own hardware scheduler that decides which kernel is to be executed on the device and monitors its execution, this scheduler can provide the most accurate time stamps for kernel execution. Such a high resolution device timer that does not interfere with the executed programs is highly desirable in a platform specific accelerator performance API.

Furthermore, some general device information can be of interest such as device memory utilization or device processor utilization. This is especially true for a shared environment where multiple applications jointly use an accelerator and, thus, also share its resources. Accelerator vendors can offer tools or APIs to query such high level information. NVIDIA for example offers the NVIDIA Management Library (NVML) which allows to poll device and memory utilization, a list of processes using the device, but also device temperatures or fan speeds. A performance monitor for an application using the accelerator can query such a high level performance API every time it is invoked to also record the general state of the device. Unfortunately, the NVML in particular only offers a very coarse grain temporal resolution for the data it makes available. The library uses data that the device driver places in the `proc` file system which in turn is only updated once per second by the OS kernel. Hence, for more precise data, a direct interface with the device driver is required.

Hardware accelerators also feature a number of hardware performance counters. These counters can provide valuable insight in order to understand whether an accelerated application utilizes the massive parallelism available on the accelerator device efficiently. GPU performance counters include metrics for bad memory utilization due to unaligned accesses or bank conflicts, but also inefficient code execution due to divergent conditional statements [AMD12, NVI12b]. As GPUs now also feature cache hierarchies, their efficient usage becomes more and more important. Including such information along with the event logs of the executed kernels allows the performance analyst to distinguish between well and not so well performing kernels. Thus, the inclusion of performance counter information also needs to be part of a specific accelerator performance API. As a summary, a performance API has to be able to report correct start and end times for accelerator kernels as well as performance counter values at these points.

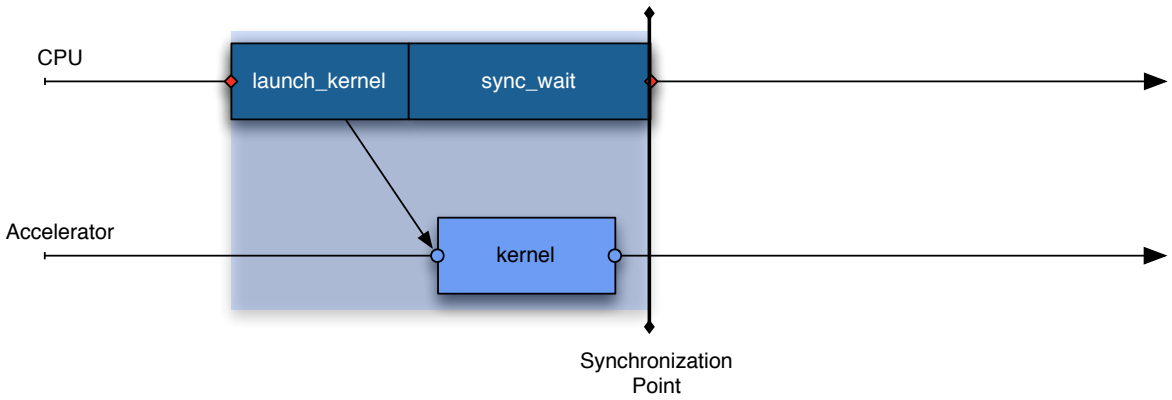

Figure 4.6: When generating time stamps for a kernel execution using forced device synchronization the overall time (red diamonds) will be augmented by the overhead to launch the kernel and to complete the wait operation for the kernel to finish. The actual kernel execution time (blue circles) is shorter. [MBS+11]

4.2.2 Methods of Acquiring the Device Data

In order to acquire the desired information from the device, the host has to have means to actually collect it. Since the only way for the host and the performance monitor to access the device is via the device driver, this driver also must be the method for accessing the desired performance data. A number of possible ways how this can be implemented are discussed in the following [MBS+11].

Forced Synchronization

The easiest way to acquire correct time stamps and performance counter values is to force the accelerator device into a synchronous execution mode. This can for example be accomplished by appending every asynchronous operation like data transfers and kernel launches with a "wait for completion" or synchronization routine. As a result only one operation can be outstanding for each device at any point in time and the start of the device operation and the end of the synchronization routine can be used as the data query point. As Figure 4.6 illustrates, the measured start and end time (red diamonds) differs from the real kernel start and end time as denoted by the cyan circles. This is the driver and device overhead for actually launching the kernel and for the synchronization on completion.

The synchronization can be forced either by the device driver or the performance monitor. The latter would automatically imply the usage of a library interception technique since the asynchronous operations have to be appended with a synchronization call. This would then be a task for the performance monitor. If the device driver were to offer a synchronous operation mode, i.e. it would automatically block the host program until the operation has finished, the accelerated program section would behave like a normal subroutine call and can be treated as such by the performance monitor. While the acquired data is—minus the obvious overhead—accurate, the obvious disadvantage of this acquisition method is the potential severe program perturbation induced by the forced synchronized execution. For programs that utilize asynchronous operations to overlap data transfers and computation, as recommended by every accelerator vendor, a better method is required.

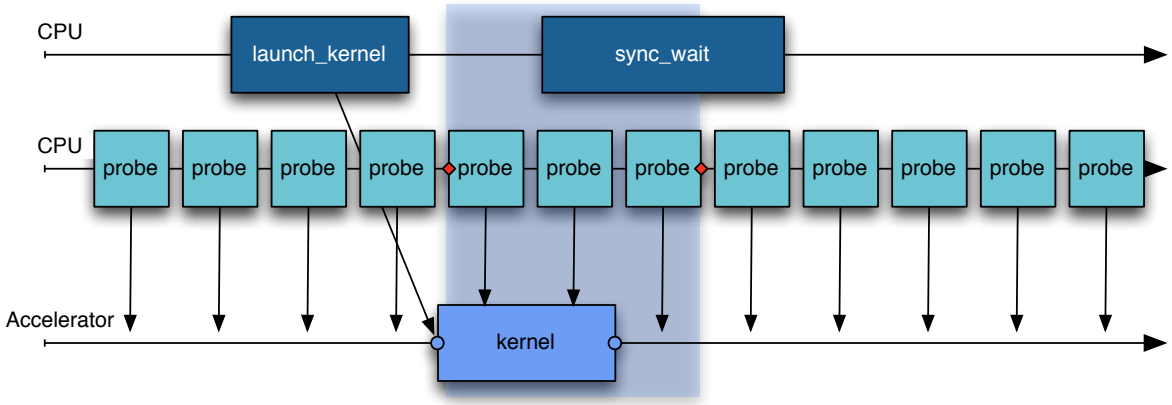

Figure 4.7: The precision of the timing of a kernel when using continuous probing is highly depending on the sampling frequency of the probing. The first probe to find the kernel running on the device will mark the start of the kernel, the first probe to find the device empty or occupied with a different kernel will mark the end of the kernel.

Continuous Probing

If the hardware accelerator driver only supports instant queries regarding the device state, i.e. which program is currently executed on the device or the value of various hardware performance counters, the performance monitor can only use a sampling based measurement approach. The performance monitor can spawn an additional thread from the host program which will continuously probe the device for data samples as shown in Figure 4.7. A new sample can be compared with the previous one to detect state changes to record as accelerator events. Such a state change can be the device changing from an idle to a busy state or vice versa as well as the execution of a new device program. Performance counter values can be tied to the recorded events on state changes, but can also be continuously recorded. Both possibilities can be implemented using OTF, since it supports counters as an individual event type. The downside of this approach is obviously the need for an additional host thread which can, if all CPU cores are already utilized by the program-under-test, severely perturb the program execution.

Callbacks

One possible method of handling asynchronous operations is to issue callbacks upon initiation and completion of asynchronous events on the device as shown in Figure 4.8. The performance monitor registers callbacks by the device driver for all device API routines that are of interest to the performance analyst. When the program-under-test invokes such a registered API routine, the driver has to ensure that the performance monitor is called at the right times, i.e. before and after the actual operation. While this is still possible for data transfers, since they are carried out by the driver, it is currently impossible for on-device events such as kernel execution. This is due to the already stated fact that a device kernel is just enqueued onto the device by the driver and the device starts the kernel whenever it is ready without notifying the driver. Furthermore, a callback mechanism typically invokes the performance monitor at the time of the API call and not at the time of the actual operation. Hence this mechanism is of benefit for

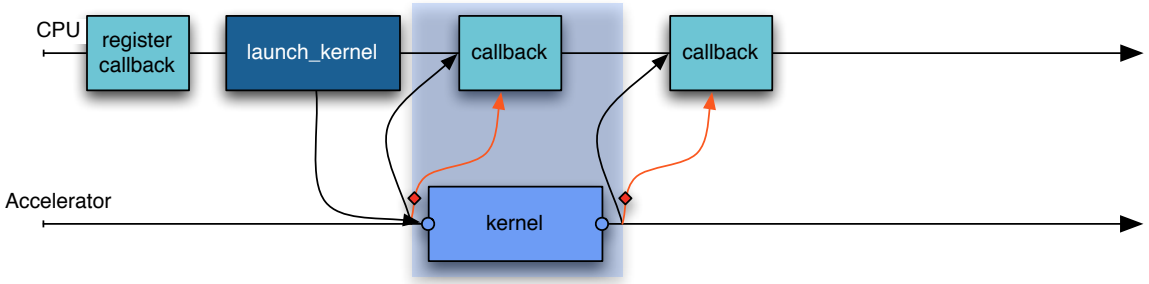

Figure 4.8: The timing of a kernel via callbacks from the device right before the kernel launch and after the kernel completion provides the highest accuracy. The callbacks provide a time stamp from the device that has to be converted by the callback handler on the host. [MBS+11]

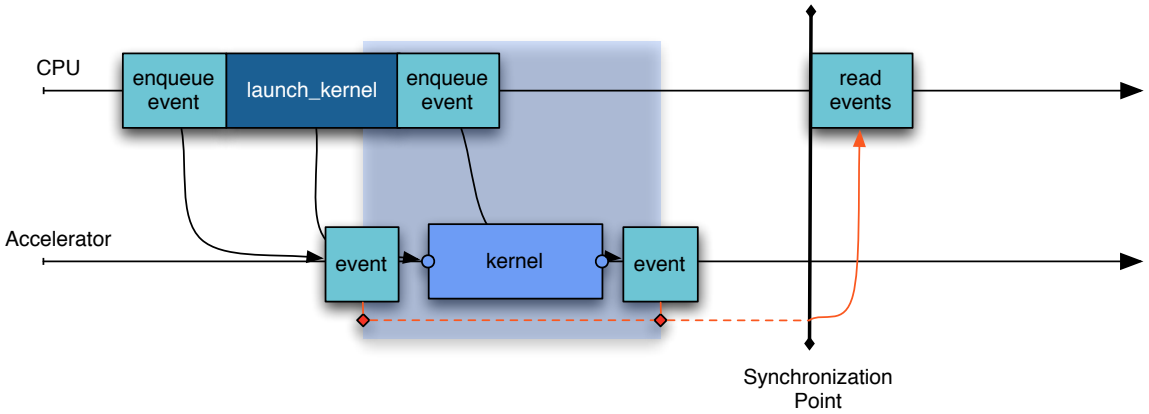

Figure 4.9: The kernel can also be timed with the previously introduced (see Section4.1.2) device events. In this case the device events have to be read back onto the host after the kernel has finished. [MBS+11]

synchronous operations since it removes the necessity for API call interception, but it does not provide the required level of detail.

Device Events

Device events are—in the case of CUDA and OpenCL—already available for timing and synchronization of device activity. The performance monitor can enqueue events into the accelerator execution stream which generate time stamps when "executed" on the device as shown in Figure 4.9. The time stamps are read back onto the host during a device synchronization phase.

In order to be of general usage the CUDA/OpenCL events have to overcome two issues: timing events must not serialize parallel execution on the device and they have to be able to also record performance counter values. Unfortunately, both is currently not possible in the desired asynchronous way for any accelerator device.

In order to combine the strengths of the various approaches the best possible course of action for a generic performance monitoring approach is the combination of all available measurement methods.

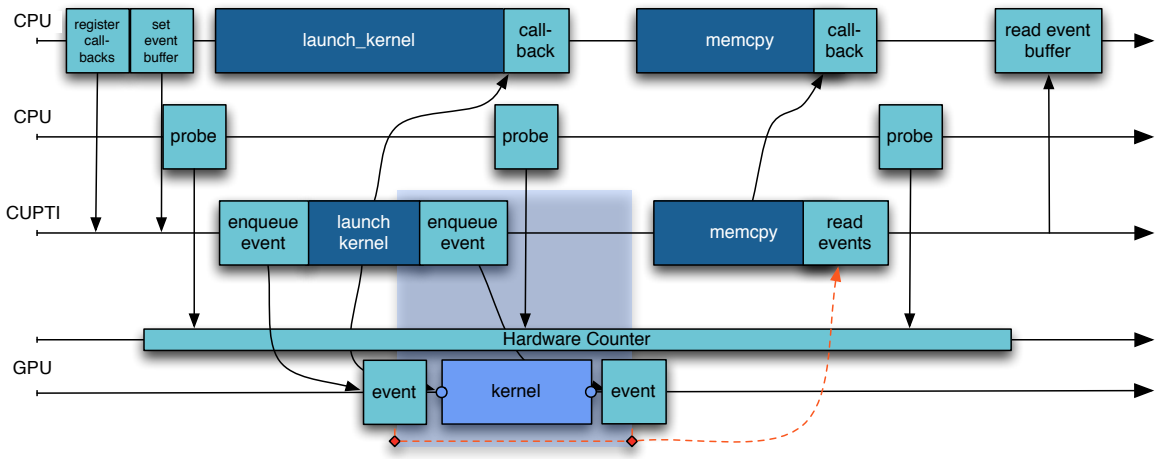

Figure 4.10: The CUPTI API provides access to all available performance data using a combination of previously described monitoring techniques. The monitor can register callbacks to CUDA API functions so that it is invoked when the application uses the CUDA libraries. Furthermore, it can probe hardware performance counters and request the CUPTI library to record time stamps for the kernel execution.

4.2.3 The Implementation of NVIDIA CUPTI

NVIDIA uses an internal tools API to provide detailed information about the usage of its GPUs to its own performance tools. As part of a collaboration with NVIDIA it was discussed how the workflow presented in the previous section and event logging in general can be mapped onto this internal API. In the process NVIDIA released the CUDA Profiling Tool Interface (CUPTI). A number of the results of this thesis have been incorporated into this interface. Due to current hardware limitations, it has to use a combination of the methods presented in Section 4.2.2 to provide the data established as a requirement in Section 4.2.1. The combination is illustrated in Figure 4.10. Performance analysis for NVIDIA CUDA based GPU accelerated applications can now be done solely using this API and does not require library wrapping anymore.

NVIDIA CUPTI consists of multiple parts to cover the various aspects of analyzing GPU performance [NVI12b].

- The **Activity API** provides event logging functionality as described in the event driven approach in the previous section. The performance monitor is responsible to initialize this capability before any other CUDA activity takes place. Furthermore, it must provide a memory buffer where the device driver will log events. Buffers must be read and if necessary rotated by the performance monitor during global device synchronization points. The activity buffers can be associated with different types of event queues: global, context, and stream queues depending on the origin of the events to be logged.

  The advantage of this API is the increased accuracy of the time stamps for the recorded events. Additionally, the activity API interfaces directly with the device's hardware scheduler and, thus, does not need to rely on timing events and in return does not suffer from the parallel stream serialization. The activity API also allows a selection which events to log into its activity buffer. The current

implementation in VampirTrace uses this API to record data transfers and kernel executions. The buffer events are translated to OTF records when the buffers is fetched from CUPTI.

- The **Callback API** allows the performance monitor to be invoked when CUDA runtime or driver routines are called by the application. It implements the callback functionality presented in the previous section. Callbacks can occur for different domains and have to be associated with one of them. Currently available are domains for runtime and driver functions, for resource tracking, and for synchronization. Each callback will receive a distinct ID to decide where it came from.

  The callback API is very useful to parse function arguments, thereby allowing to determine data transfer sizes. One disadvantage of this approach is that only one callback occurs when entering the subscribed routine. Hence, it cannot be used for timing the end of the host's API activity.

- The **Event API** is used to access hardware performance counters on the GPU. The number and types of available events differs from one GPU generation to another, but can be queried using this API. Some events can be collected concurrently by placing them in an event group. Events can be collected continuously or just during the execution of a kernel. The performance monitor has to subscribe to an event prior to starting a kernel and read the counter values after it has completed. Callbacks are available to ensure the monitor is called at the right times. If one uses the continuous counter collection mode with an additional sampling thread, the performance monitor can also generate counter values during the execution of the kernel. The event API, thus, utilizes both the callback and the continuous probing approach from the previous section.

  The available events also differ with respect to which hardware part they cover. Counters can be restricted to streaming multiprocessors (SM), the texture processing unit (TPC), or the memory subsystem (FB). Counters for the SMs are further grouped into different domains. Depending on that domain they are available for all SMs, thus, providing a grand total for the device, or just some but not all of the SMs. This places a high burden on the performance analyst with respect to interpreting the counter values correctly.

- The **Metric API** abstracts and aggregates specific events to more convenient performance metrics. Metrics are subscribed to in the same manner as events. The metric API automatically normalizes events that are not available for the whole device, e.g. all SMs, so that it appears that they were. As a result the kernel under test must load the device evenly to provide a useful measurement.

The integration of CUPTI into VampirTrace to record CUDA GPU activity with very low overhead at a high precision utilizes all four sub-APIs concurrently. Obviously the Activity API is used to generate the time stamps for the device related events. Callbacks are used to track the device utilization in general, so that a program establishing a context with the device automatically leads to the creation of an event log for this context and the assignment of an activity buffer as well. Furthermore, callbacks are used to track whether the program-under-test uses multiple execution streams and to track them concurrently with own event logs and activity buffers. Callbacks for data transfers are used to record its parameters (size as well as the origin and destination). The former is important for calculating the transfer bandwidth during the event data analysis, the latter is necessary to provide information about which stream the transferred data belongs to. Events and metrics are used to provide additional data about the executed kernel. This allows the performance analyst to understand not only quantitative aspects of the kernel execution—how

Table 4.2: The Monitoring Overhead for the CUDA Runtime API using CUPTI is measured measuring 100 times 1000 repetitions of each event using the CUDA Toolkit version 4.2 and the CUDA Driver version 295.41.

GPU	Event	Execution Time [ms]			Overhead per Event [μs]			Flush
Type	Type	min	avg	max	min	avg	max	[μs]
S1070	Kernel	8.381	8.393	8.438	2.916	**2.925**	2.958	277.215
S1070	Memcpy	117.269	131.734	170.914	0.337	**3.491**	5.128	346.25
S2050	Kernel	7.343	7.889	22.473	5.019	**5.097**	7.835	266.215
S2050	Memcpy	27.631	28.350	46.843	8.054	**8.247**	15.643	336.86

often are various kernels invoked and how long do they run— but also qualitative aspects—how well they utilize the device.

The overhead for using CUPTI for monitoring CUDA activity is shown in Table 4.2. The flush operation which reads the time stamps from the CUPTI activity buffer is only called once per 1000 iterations and has a constant overhead. Since the execution times vary quite a bit, the best approach to determine an overhead is to use the average. These numbers already expose a trend: kernel invocations induce a smaller overhead than asynchronous memory transfers and monitoring on the newer Fermi GPUs (S2050) has a lower overhead than on the older Teslas (S1070). Furthermore, the overhead when using the CUPTI API is much lower compared to the event based method shown in Table 4.1.

4.3 Accelerator Based Event Logging

After discussing how information about the accelerator usage can be extracted by monitoring host activity, the focus of this section is on monitoring program execution directly on the device. Doing so is at best difficult and at worst impossible depending on the accelerator device under investigation. One example where accelerator program monitoring is virtually impossible are FPGAs. Since an FPGA program actually programs the hardware, it would have to be modified to accomodate performance monitors. While this is theoretically possible, this modification would require a lot of time and might even render the program unusable because the added monitoring hardware might oversubscribe the available resources. Nevertheless, for the case where monitoring is possible three different approaches for acquiring performance data are discussed in the following.

4.3.1 Accessing Performance Counter Information

Hardware performance counters, as mentioned briefly in the previous section, are a well established method for acquiring detailed information about hardware activity during the execution of the program-under-test. They are available for almost any device in a modern computer from CPUs to mainboard chipsets or extension cards like network adapters. While most hardware counters have their origin in the actual hardware design where they are used for ensuring that the device-under-test is working properly, the same information can also be of great interest to the performance analyst since it provides insight into how well the program is utilizing the hardware device.

The main difficulty in the context of hardware accelerators and hardware performance counters is the inability of the accelerator device to access the counters. Aside from heterogeneous processors like the IBM Cell/B.E. where the SPUs can access their own performance counters, accelerator performance counters are only available to the host via the PCI-Express link to the device. As a result, device programs cannot use this information directly, e.g. for automatic device adaption. Furthermore, querying the counters from the host involves a PCI-E transfer which, if done too frequently, will severely perturb the accelerator program execution, given its execution frequently relies on data transfers over the same link.

On synchronously operating platforms like CPUs, an event based performance monitor will query the performance counters for every event and, thus, can provide counter differences for each program phase or subroutine execution. For accelerators this is more complicated due to the asynchronous execution nature on these devices. A performance counter acquired with the launch of an accelerator program only provides data that is actually correlated with this program if the device is running in a forced synchronization mode. In all other cases the counter sample could be from another program running before the launched program. Since callbacks for the start of the accelerator program do not exist—in this case the callback could be used to read the performance counter values at that point—and device events can only record time stamps, the remaining possibility for accurate performance counter data is continuous probing. In this case an additional host thread is used to periodically poll the subscribed performance counters. The frequency of this polling needs to be chosen carefully and is also application dependent in order to keep the program perturbation minimal. As mentioned before a very high sampling rate will basically block the PCI-E bus between the host and the device and, thus, significantly throttle data transfers over the same link. At the same time a too low frequency might not catch the begin and end of an accelerator program to provide a meaningful measurement.

The PAPI performance counter API and the VampirTrace performance monitor have been extended to provide access to CUDA device performance counters [MBS+11]. The PAPI CUDA component provides a direct interface to the CUPTI event API. A benefit is that a performance analyst who knows the PAPI counter API does not have to deal with the CUPTI event API, but can simply use the well established PAPI interface for querying GPU performance data. VampirTrace supports querying performance counters for every GPU related event. This is typically a callback from the CUPTI callback API which invokes the VampirTrace CUDA monitor. If performance counters are monitored, their current values will be read using the CUPTI event API. While the performance counters are not available directly on the GPU, they can be sampled from the host and provide a similar level of detail as is available for host programs.

4.3.2 Kernel Wrapping

An obvious flaw of the currently available device timing events is their inability to hold any additional information like performance counter values. One possibility to create such extended events would be to replace them by performance monitoring kernels. These kernels log a time stamp together with meta information about its context and potential performance counter values directly on the device. The performance monitor on the host does in this case not surround the accelerator program by timing events but by the measurement kernels, so that the first one logs an enter event and the second one a leave event. This approach implements event logging directly on the accelerator device.

In order to log the performance data directly, the performance monitor running on the device requires its own log buffers in the device memory. This can be difficult since all accelerators are very limited in this resource and as a result only very small buffers can be used for performance events. Furthermore, the performance data needs to be moved to the host to be transferred into the event log of the overall program execution. This can be done by a typical double buffering strategy where the host is transferring one buffer while the device's performance monitor is currently filling the other. A buffer switch can be done before or after a synchronous operation on the device since the host and device performance monitor will be in sync at this point.

There are two prerequisites for such an approach: first, performance data has to be persistent over accelerator program invocations, meaning that the clock signal used for time stamps runs evenly and performance counters are not reset in between accelerator programs. This can be ensured for the studied accelerators and does not pose a limitation. Secondly, a device program has to have access to all required data to generate the correct event log entry. Unfortunately, this is only the case for the clock signal and the meta information, but not the performance counter data—as mentioned previously. As a result this approach has not been implemented yet, since it provides no benefit over the already available host based measurement approaches. It could, however, become an option when accelerators also provide performance counter data directly to the device.

4.3.3 Kernel Instrumentation

All previous methods of event monitoring—with the exception of continuous probing which in turn is not really an event based but a sample based measurement method—are focused on recording the correct start and end times for programs executed on an accelerator. Even this, from a perspective of a non-accelerator programmer, rather trivial task proves to be quite complicated due to the asynchronous execution nature of the accelerator. If the temporal resolution of the performance monitor was to be increased to also include events generated during the execution of an accelerated program, manual instrumentation as introduced at the beginning of this chapter could be applied. This approach is somewhat similar to the just mentioned kernel-wrapping: an event trigger routine which can also be called from the accelerated program is provided by the performance monitor. As a result, this method can show the internal structure of the kernel and how it is mapped onto the accelerator, since each thread of the kernel can now produce a time stamp. In case of CUDA or OpenCL this is a device function which will read the device clock and write this time stamp together with its context into a device held event buffer. Especially on GPUs which execute thousands of threads concurrently such an approach can be difficult, since also generates thousand of events at the same time. At this point, data aggregation directly on the device is inevitable.

In order to actually gain insight into a kernel in a (semi-)automated fashion, one requires a compiler that can inject callbacks to a performance monitor also for accelerated code as it is available with compiler instrumentation for CPU code. Unfortunately, no accelerator compiler does yet support this performance analysis technique. As a last resort one can analyze and modify the generated binary code [Hö12]. For OpenCL this approach is unusable since the OpenCL driver compiles the program prior to its execution and the generated binary code for the accelerator device is typically undocumented by the vendor. CUDA uses Parallel Thread eXecution (PTX) code as an intermediate language for its accelerator programs.

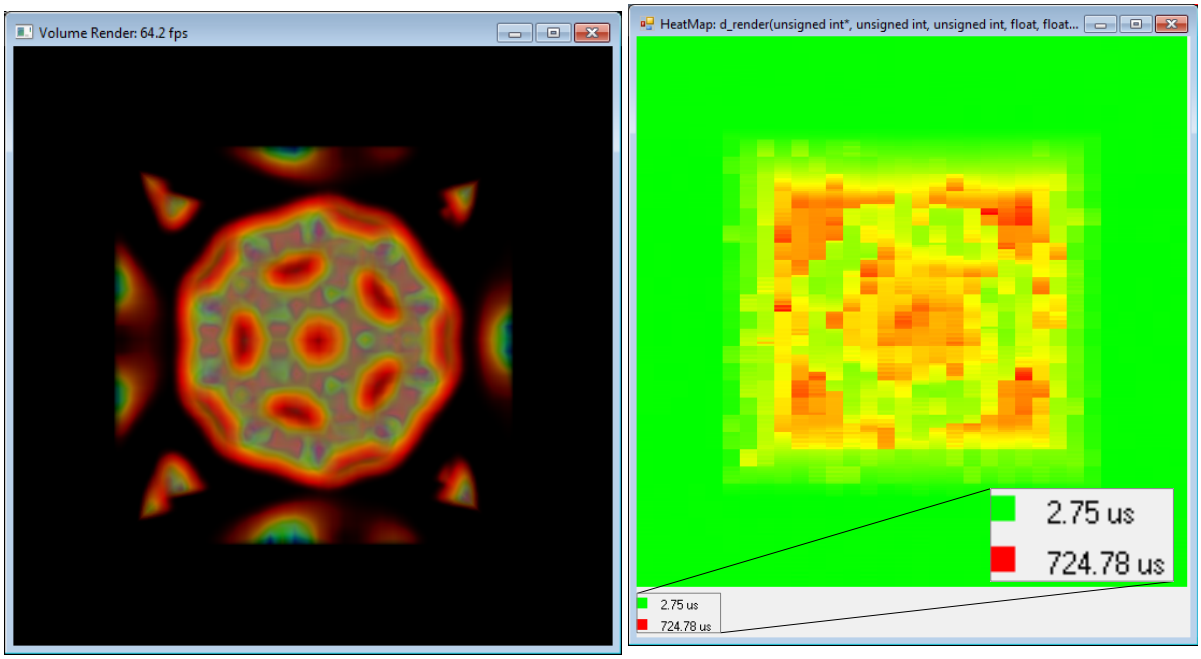

Figure 4.11: The side by side view of the actual output of the volume rendering kernel and the thread block heat map of the used grid. It can be seen that the execution times of the thread blocks in the black areas, where no computation is needed, is low, while it is high in the parts where actual rendering needs to be done. [Hö12]

When a CUDA accelerated application is executed the driver translates the PTX code to device specific assembler (SASS) just-in-time. GPU Ocelot [KDY11] can modify the PTX code of a CUDA application in order to insert the callbacks to a performance monitor as described before for a manual instrumentation approach. Doing so, however, still requires the performance analyst to modify the assembly code, albeit with the help of a tool.

In order to extend event logging to also capture such device internal activity the instrumentation process needs to be automated which was investigated together with NVIDIA. The work was able to use the internal tools API from NVIDIA directly [Hö12]. This API offers a method for code injection or modification after the SASS code has been generated but before the binary is loaded onto the device for execution. Debuggers use a similar approach to introduce debug symbols and handlers into the code. One can for example substitute every `exit` symbol in the code with a jump to a self provided subroutine which will read the clock signal and log it into a specific memory location before executing the original `exit` command. As a result one receives a time stamp for the kernel completion for every thread, warp, or thread block depending on how the memory location is chosen. The beginning of a kernel can be recorded in a similar fashion. The recorded kernel durations can then be plotted along the same distribution scheme as used for the execution to see where the execution took longer. Figure 4.11 highlights this with a very intuitive example.

The same injection approach can be used to provide an instruction counter for specific instructions on the device by emulating a hardware counter. NVIDIA GPUs for example still lack a counter for executed floating point instructions. At the same time this is the basis for a very important metric for device usage efficiency: floating point operations per second (FLOPS). In order to count all floating point operations

the developed injection library substitutes all floating point operations with a jump to a subroutine that increases a counter by one, executes the original floating point operation, and returns to the original program location. The thread-wise counters are reduced to a warp, thread block, or global sum which can then be used to calculate floating point throughput. It is obvious that especially this technique produces a significant execution overhead, but currently is the only possibility to provide an exact count for the number of executed floating point operations. A repeated execution of the program-under-test, which counts the floating point operations in the first run and measures the execution time in the second run, Due to the proprietary nature of the used tools API, this work is not available publicly. It is, however, part of NVIDIA's Parallel NSight tool.

4.4 Effects of Overlapping Metrics

As it was already established in Section 4.1.2 the event log generator has to reuse existing metrics to record the additional information available from monitoring the hardware accelerator usage. This is a limitation of the used file format which does not support accelerators per se. Hence, the presented solution "reuses" metrics for multiple purposes: accelerator execution streams are represented like CPU threads. Their event data will be placed in event streams that have a child relationship to the host processes event stream. Additionally, data transfers between the host and device can be recorded the same way as inter-process communication would be. The data presentation of the event log data is also unprepared for the additional level of parallelism. Yet at the same time, it is highly desirable to reuse most of the existing infrastructure to reap the benefits of their already established scalability and stability. As a result the timeline visualization of such an event log can preserve this parent-child relationship and display the accelerator events right next to the data from the host process, as already shown in Figure 4.5. The data transfers are also included in the display, they are represented by the black lines connecting the two communication partners—in this case the host and accelerator device.

The "reused" or rather overlapping metrics lead to undesirable side effects in case of programs-under-test that use more than one parallelization paradigm for example multiple host processes communicating with MPI but also using hardware accelerators. In this case the metric "message" is used with two different meanings: MPI messages and host-to-device data transfers. As a result all message related profiles are a combination of the two metrics and the message lines drawn into the timeline display can also stand for both. While one could argue that for the timeline display this is not much of a concern since the source/destination indicates what kind of message was exchanged, this argument is invalid for highly parallel event logs where there is not enough room for labeling each event stream. Thus, distinguishing accelerator from host process event streams is more difficult. Event logs of highly parallel hybrid applications using thousands of concurrent event streams also create another problem: in order to display a timeline of the whole event log some event streams need to be omitted and the event streams chosen for presentation are only one pixel high. This is a highly useful approach for programs-under-test using just one parallelization paradigm, since the aliasing still preserves the general overall behavior of the program as shown in Figure 4.12. If the event log now also holds event streams of another parallelization paradigm, the aliased global timeline display—as shown in Figure 4.13—will be "polluted" by those event streams with a different content, thus, disturbing the otherwise still visible patterns in the aliased display.

Figure 4.12: The Vampir Master Timeline display for an event log of the S3D application running on over 200,000 processes on Jaguar shows that even this highly aliased display maintains the overall program behavior [ISC+12].

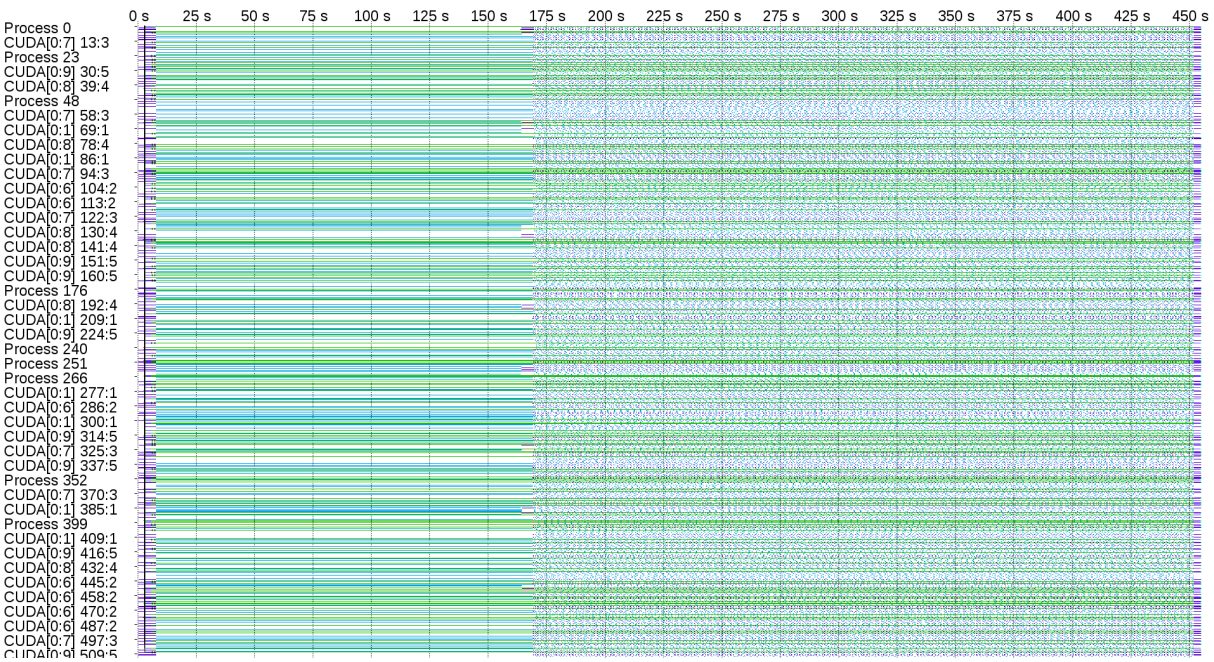

Figure 4.13: The Vampir Master Timeline for an application using MPI, CUDA, pthreads, and OpenMP on 512 compute nodes with 512 GPUs shows how the aliasing between the event streams of various parallelism layers turns the display basically into colored noise.

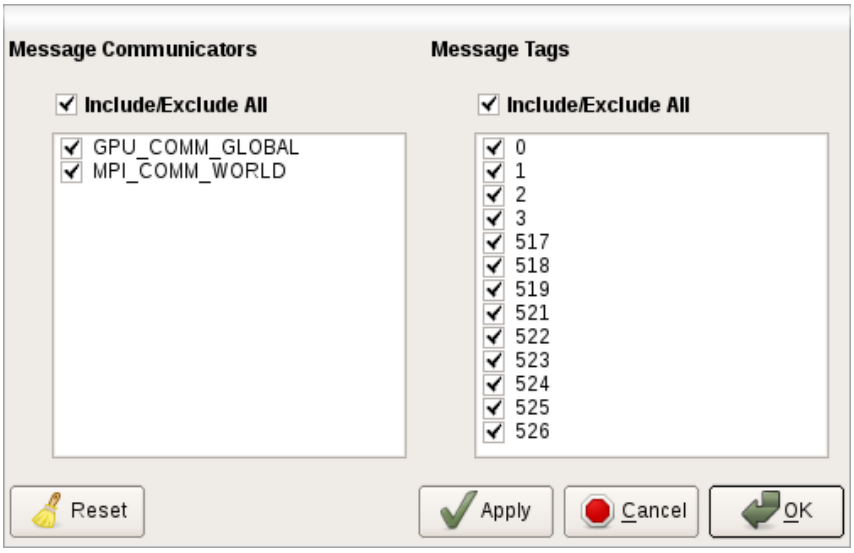

Figure 4.14: The Vampir Message Filter dialog presents the used message communicators and tags and allows a selection of which messages to display. Since the host-to-device transfers are logged using an own artificial communicator, they are selectable as a group. MPI messages can for example now be turned off to profile the GPU memory transfers. All message related statistics show then only GPU related message data.

Most interestingly, this is already true for "classic" hybrid applications using MPI communication for inter-node communication and threads within a node but has not yet been addressed as an issue. A possible reason might be that the host processes, i.e. the MPI processes, of such hybrid applications typically participate in the multi-threaded regions as well which still provides are largely homogeneous timeline display. The resulting performance profiles were, as a result, still useful. When using accelerators where host and device follow disjoint execution paths, the host and device event logs are different. Generating performance profiles, as an event log visualization tools would typically do, combines performance data that is not directly comparable into one profile. The performance analyst instead might want to profile the host and device individually to understand the performance bottlenecks of each parallelization layer itself before studying performance problems due to their interaction.

In order to separate the fused profiles for threads and messages the performance monitor has to add additional information to the event log file. In case of the data transfers between the host and the device this is an additional, artificial communicator that is associated with all such transfers. The Vampir visualization provides a filter mechanism to only regard certain message types as shown in Figure 4.14. As a result one can turn the inclusion of MPI messages and accelerator data transfers on and off to profile them individually. After doing so the capabilities of Vampir can be used to locate bad data transfer behavior of either kind. The same is true for the global timeline display. Here a process filter—as shown in Figure 4.15—can be applied to present only data from the accelerator or host event streams or a combination of both. This allows to profile each parallelization layer individually in order to locate performance bottlenecks. At the same time a view of all event streams enables the performance analyst to study how well the interaction between the used parallelization layers is implemented.

A straightforward approach to solve the aliasing problem for the global timeline display is to display the event streams for each parellelization layer in an own timeline display. This does, however, not solve

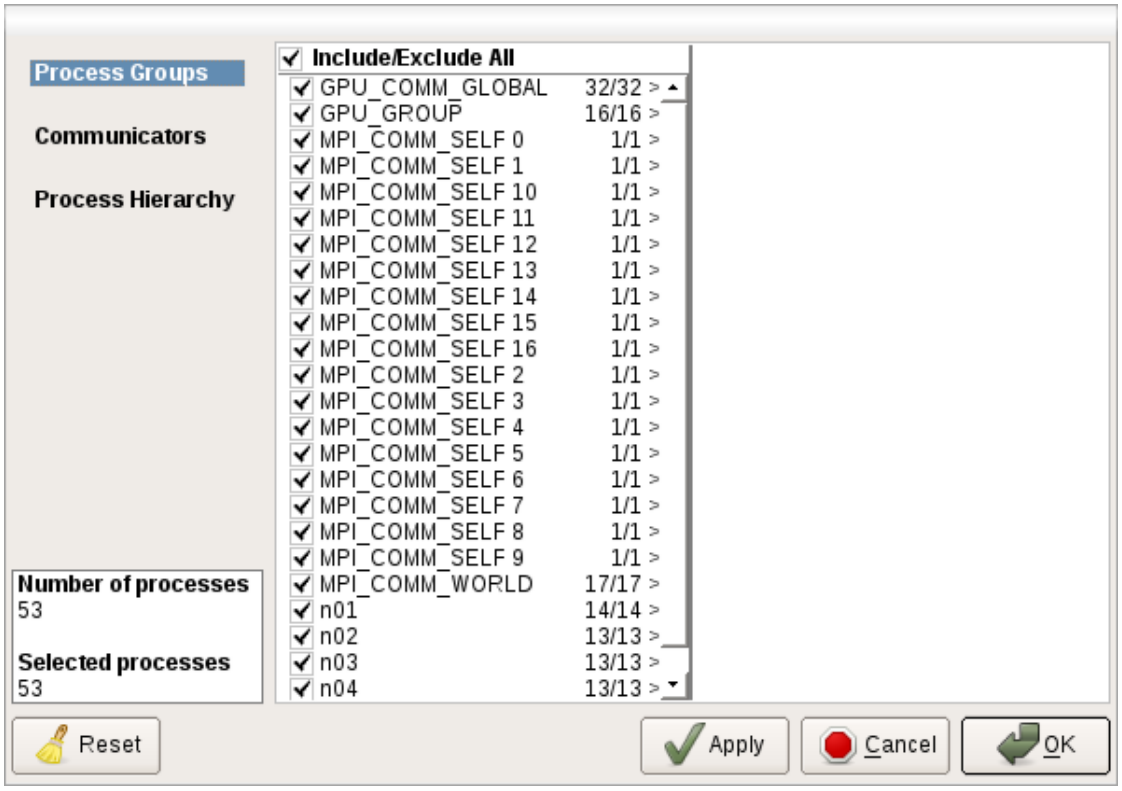

Figure 4.15: The Vampir Process Filter dialog allows the selection of the recorded process groups. In this case the performance monitor has added the GPU_GROUP and the GPU_COMM_GLOBAL groups to the existing MPI related process groups. The GPU_GROUP contains only the GPU activity event streams. If this is selected, the Master Timeline will provide again a useful aliased data presentation—in this case of the GPU event streams.

the issue with the fused performance profiles. These displays need to be available for each overlapping metric separately as well, thus, limiting the scarce resource "display space" even further. As a result one will probably not look at all displays at the same time and return to the same mechanism as provided with the previously introduced filtering.

4.5 Parallel Program Flow Graphs

The strength of a timeline display of event log data leads to a major weakness as well: presenting data for a lot of different event streams or large measurement periods can only present a limited data set. The user has to locate interesting portions in the presented data and has to enlarge this region in the timeline display manually when using a visualization tool like Vampir. At the same time, a typical indicator for a performance problem can be described quite specifically: locate event streams that behave differently from the majority of the other streams. This "difference" can be of a structural or temporal nature. For example, one process follows a different execution flow such as a master process or one process calling the same subroutines but taking longer to complete one of them. The latter example results in an offset in the timeline representation of that process. The performance analyst has to locate the source of the offset, i.e. the one subroutine that took longer than the others, to find the cause of the

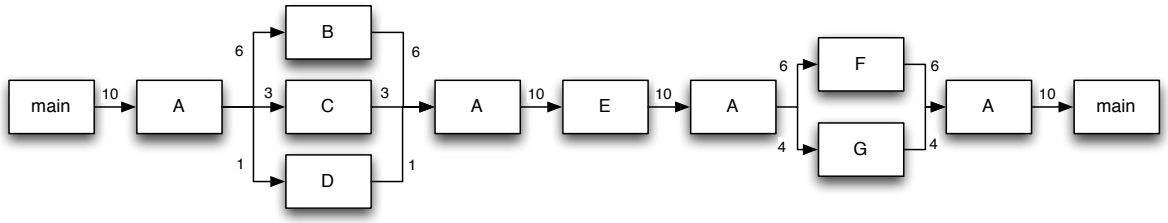

Figure 4.16: The parallel program flow graph (PPFG) fuses the state transitions from multiple event streams into one graph. In this example the states “main”, “A”, and “E” are shared by all ten event streams. The states “B”, “C”, “D”, “F”, and “G” are only entered by one to six event streams.

imbalance in the parallel execution. The alignment of the performance events along their global time stamps provides a valuable overview how the processes interact but can complicate the detection of the cause of performance problems, especially for very large event logs. Parallel Program Flow Graphs (PPFGs) can overcome these drawbacks by substituting the events with program states and the global time stamps with state durations.

The main idea of a PPFG is to use the C3G memory structure as introduced in Section 3.3 as a basis for a graph based visualization. The goal is to present an event log as a series of program states along with the transitions from one state to another as shown in Figure 4.16. The C3G data structure offers methods to determine whether the graphs or subgraphs are structurally identical. It can join nodes of subgraphs from two event streams when they have an identical structure and the duration of the program states in the subgraph is within the margins set for the creation of the graph. Furthermore, the C3G provides a structure map that determines identical structures in the subgraphs [Knü12]. Two event streams with the same sequence of program states can then be visualized with the same graph. If done for a whole event log, this will reveal all occurring program flows of the application. Furthermore, it also separates the various layers of parallelism since they each follow a different execution path.

4.5.1 Offline Construction of a PPFG

The goal of the offline construction of a PPFG is to construct the complete graph and store it for later interactive analysis. The graph is constructed by pairwise comparisons of C3Gs of event streams. The comparison is aided by the usage of genome sequence alignment techniques [SW81].

High-Level Sequencing In a first step the algorithm identifies nodes that are identical between the two C3Gs. This is done by first coloring one graph G_1 and traversing the other G_2 in-order. Once this traversal hits a colored node, a perfect match between the two subgraphs is found. The node ID is recorded for G_2. The in-order traversal of the subgraph skips the children of the current node and immediately returns to the parent since it is a intrinsic property of the C3G that all children of a colored node are also colored. After traversing all nodes of G_2 the recorded node IDs form a sequence S_2. Next, the order is reversed: G_2 is colored and G_1 is traversed, resulting in a sequence S_1. The sequences S_1 and S_2 contain all program states that are shared between the two event streams although not necessarily in the same order. Thus, one can apply a genome sequence alignment algorithm allowing gaps and

mismatches to find the best alignment between the two sequences. The begin and end of the two event streams also serves as a perfect match to provide a boundary for the algorithm.

The benefits of applying this high-level step are twofold. First the event streams are dynamically split at points where they perfectly match regardless of the stack level. This provides a more flexible approach than previous combinations of genome sequencing and event logs. Furthermore, the C3G matches can actually be not only single program states but a whole subgraph in the C3G. All children of a matched C3G node automatically also provide perfect matches and do not have to be considered.

Low-Level Sequencing The next step is the expansion of the states between the matched states from the high-level sequencing. This step provides two sequences S_1 and S_2 that need to be aligned as well. Repeating this for the next perfect matches until the end of the event stream generates an overall sequence mapping of the two event streams which in turn can be presented as the PPFG. To do so the two genome sequences are traversed: a match is represented as a joint state, a mismatch as two states, and a gap as a transition skipping one node as shown in Figure 4.17.

Complexity When aligning two sequences the longest sequence determines the overall complexity. If the longest sequence has n states, then the complexity of the alignment will be $O(n^2)$. As already mentioned in Section 3.3, this is impractical for real world event logs which easily contain millions of states per event stream. The proposed two staged approach reduces this complexity. Only in the worst-case scenario where no match is found in the high-level sequencing, the complexity is the same as for the flat alignment.

The high-level sequencing locates a matches in the C3G data structures of the two compared event streams. The complexity of this step is $O(n)$ for the coloring plus $O(a^2)$, where a is typically much smaller than n:

$$O(n + a^2) \quad | \, a \ll n \tag{4.1}$$

The low-level sequencing has to match a sub-sequences. The length of those sub-sequences is b_i states, where

$$\sum_{i=0}^{a} b_i \leq n \tag{4.2}$$

Since the matched states in the high-level sequencing can actually be nodes with sub-states, the overall number of states that have to be aligned in the low-level sequencing is usually lower than n. One can assume that all sub-sequences are similar in length ($b_i = b$). The complexity of mapping one sub-sequence is $O(b^2)$, the overall complexity of the low level sequencing is then

$$O(a \cdot b^2) \quad | \, a \cdot b \leq n \tag{4.3}$$

This leads to a complexity for the whole process of comparing two event streams of

$$O(n + a^2 + a \cdot b^2) \quad | \, a \ll n \wedge a \cdot b \leq n \tag{4.4}$$

This time complexity can be reduced, since all a alignments in the low-level sequencing can be done in parallel. The overall complexity is then $O(n + a^2 + b^2)$.

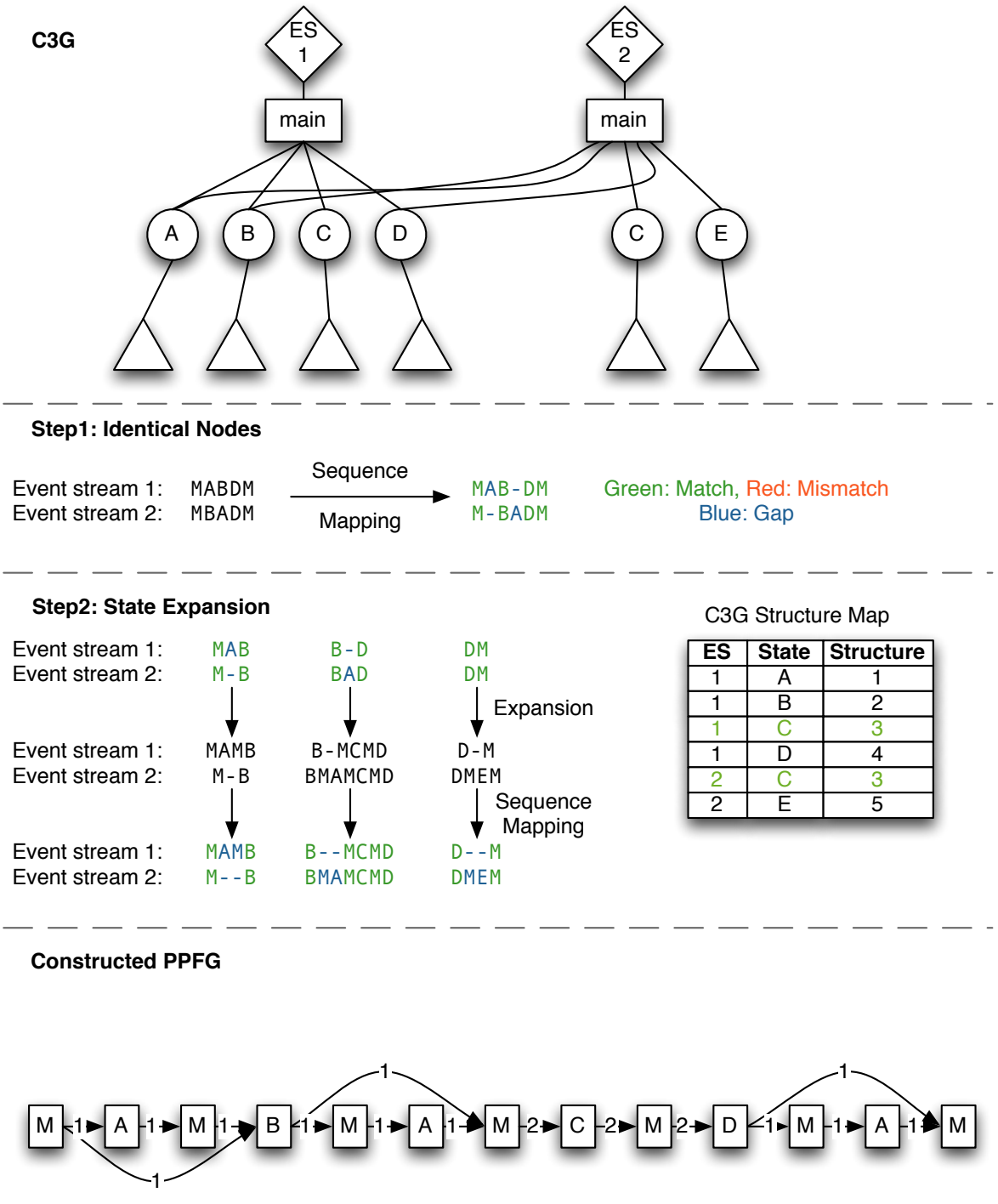

Figure 4.17: The generation of a PPFG for two event streams uses a two staged approach to divide the potentially very long event stream into smaller chunks. In a first step all direct matches from the C3G are sequence aligned. In the second step the missing states between the matches from step 1 are expanded. Then the C3G structure map is used to find more identical states. The generated sequence is aligned as well and converted into a graph. The process is repeated for all pairs of event streams.

Global Sequencing A complete PPFG requires $\frac{k^2}{2}$ pairwise comparisons for all possible pairs. The pairwise comparisons, however, can be done in parallel.

In order to further reduce the time needed, k is reduced by clustering similar event streams. The similarity can be directly derived from the sequence alignment. The alignment algorithm produces the alignment of two sequences as well as a score for this alignment. The score is determined by assigning individual scores to certain events for the alignment. In this case a match is scored with 0, a gap with -1, and a mismatch with -2. The best possible score for aligning two identical sequences is then 0. Two program flow graphs G_1 and G_2 can be clustered into a joint graph G_c if their similarity s exceeds a predefined threshold q.

$$G_c = \begin{cases} G_1 & \Leftrightarrow |G_1| \geq |G_2| \\ G_2 & \Leftrightarrow |G_1| < |G_2| \end{cases} \wedge s_{G_1,G_2} > q \tag{4.5}$$

The similarities can be stored in a matrix S.

$$S = \begin{pmatrix} 1 & s_{G_1,G_2} & s_{G_1,G_3} & s_{G_1,G_4} & \cdots & s_{G_1,G_m} \\ 0 & 1 & s_{G_2,G_3} & s_{G_2,G_4} & \cdots & s_{G_2,G_m} \\ 0 & 0 & 1 & s_{G_3,G_4} & \cdots & s_{G_3,G_m} \\ 0 & 0 & 0 & 1 & \cdots & s_{G_4,G_m} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 \end{pmatrix} \tag{4.6}$$

As a representative for the cluster the longest event stream is chosen. The following pairwise comparisons are only needed against the cluster. Hence every found cluster saves $\frac{k}{2}$ pairwise comparisons.

In order to provide a hint to temporal divergence in the execution of the same program structure, the duration information for each program state is stored with each node in the PPFG. Nodes used jointly store a minimum, average, and maximum duration. As a result states with very high runtime deviations can easily be determined.

4.5.2 Online Construction of a PPFG

The idea behind the online construction of a PPFG is to convert C3G data on the fly into a visualization that depicts the program flow recorded in an event log. The proposed algorithm for this is similar to one proposed for the offline generation of a PPFG, but it is much more sensitive to the number of program states to compare. Hence, the online algorithm starts from one node in the C3G and only considers program states on the next call stack level. This reduces the number of program states to compare dramatically, since the number of callees is typically limited.

The program flow is now recorded as a genome sequence and a pairwise sequence matching like in step 2 of the offline algorithm is used to construct the PPFG of an event stream pair. The same clustering can also be applied to reduce the number of pairwise comparisons which still depend on the number of event streams in the event log.

The comparison of states determines whether two states from different event streams are structurally identical. This is done by using the structure map of the C3G. In case the nodes are structurally identical

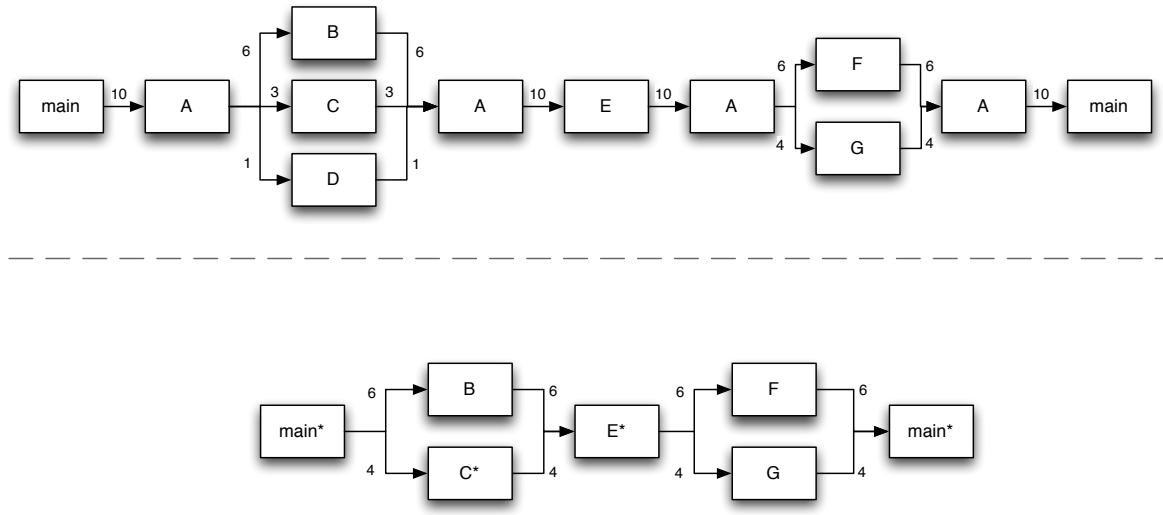

Figure 4.18: A PPFG can be compressed by joining the nodes of short durations with their neighbor and fusing nodes representing few event streams. In this case the state “B” has more members than “C” and “D”, thus, “C” and “D” are joined into “C*”. The temporal compression works in the same manner: the states of “A” around “E” and “main” are rather short. Hence, they are fused with their neighboring states to “E*” and “main*”.

they will be joined in the PPFG and their durations will be merged with the minimum, average, and maximum stored in the node. In case of a mismatch, the nodes can still have the same name, but their underlying structure is different. In order to provide a measure, how similar or different the nodes are the similarity of the nodes children is determined by the low-level sequencing approach from the offline PPFG generation.

The generated PPFG is then presented to the performance analyst who can select a new root node to repeat the algorithm on the next call stack level. Thus, it is possible to gradually resolve structural and temporal deviations in the program execution by zooming into execution flows with a high runtime difference or low similarity.

4.5.3 Display Aliasing of a PPFG

For both the offline and online generated PPFG, the available display area can be too small to draw the complete graph. Hence, a graph aliasing simliar to the timeline aliasing in Figure 4.12 is needed. The number of nodes that can be fitted into the available display area in both the horizontal and vertical dimension can be derived directly due to the fixed display size of each graph node. The compression of the displayed graph in the horizontal direction is done by joining program states by duration, i.e. join nodes with short durations first. The compression in the vertical direction is done by combining nodes by number of members, i.e. join nodes representing the fewest execution streams first. The principle is illustrated in Figure 4.18.

Another possibility to include more information in the limited display space is to encode the similarity of the remaining states into a color gradient from green over yellow to red. Green stands for 100% similarity,

yellow for 50%, and red for 0%. The background of the nodes is then colored respectively. Nodes representing the same structure are of course green. However, when they are fused with neighboring nodes for aliasing purposes their similarity decreases as well.

As a result the compressed PPFG shows the most prominent program states of the underlying event log. "Prominent" in this case means states where most time is spend and which are present in most event streams. Interactive navigation on the graph allows first to display additional information about a single node, such as representing event streams as well as minimum, average and maximum duration of the state. Furthermore, a node can be expanded so that the next zoom level uses all available display space to present the fused content of the node. This is similar to the navigation in a timeline.

4.5.4 Other Presentation Approaches Using the PPFG Data

In the construction process of a PPFG the similarity of one event stream with respect to all others is determined. The event streams or clusters from the generated similarity matrix S (Equation 4.6) can be graphed as a radial chart. Using one event stream as the origin, the other event streams can be aligned as dots with a distance from the center according to their similarity to the selected event stream. Similar event streams are placed close to the center of circular display, not so similar ones further away from the center. The additional room in the circle can be used to place the dots representing the event streams according to their respective similarity as well. Hence, if two similar event streams that are not like the currently selected origin need to be placed, they will be drawn right next to one another. The interactive analysis can display information about each cluster or event stream by clicking on it but also allows to use the currently selected cluster or dot as the new origin. As a result the performance analyst can gain an understanding of the various types of event streams contained in the underlying event log as shown in Figure 4.19.

4.5.5 Using PPFG for Event Logs from Hybrid Applications

Another strength of PPFGs is their ability to seamlessly integrate program states from different parallelization layers. A thread that is generated by the program-under-test at some point during the program execution has a parent state, for example the state `pthread_create`. This state then has two following states, one for the next parent activity and one for the newly started thread. The principle is illustrated in Figure 4.20.

The usage of an accelerator can be visualized in the same manner. The accelerator `launch` command is the parent for the accelerator states. The divergent flows are combined when the host reaches a synchronization state. One difference to host threads is that more work can be added asynchronously prior to the synchronization. The relationship between these following kernel launches and their corresponding device activity can be preserved by using dashed lines as shown in Figure 4.21.

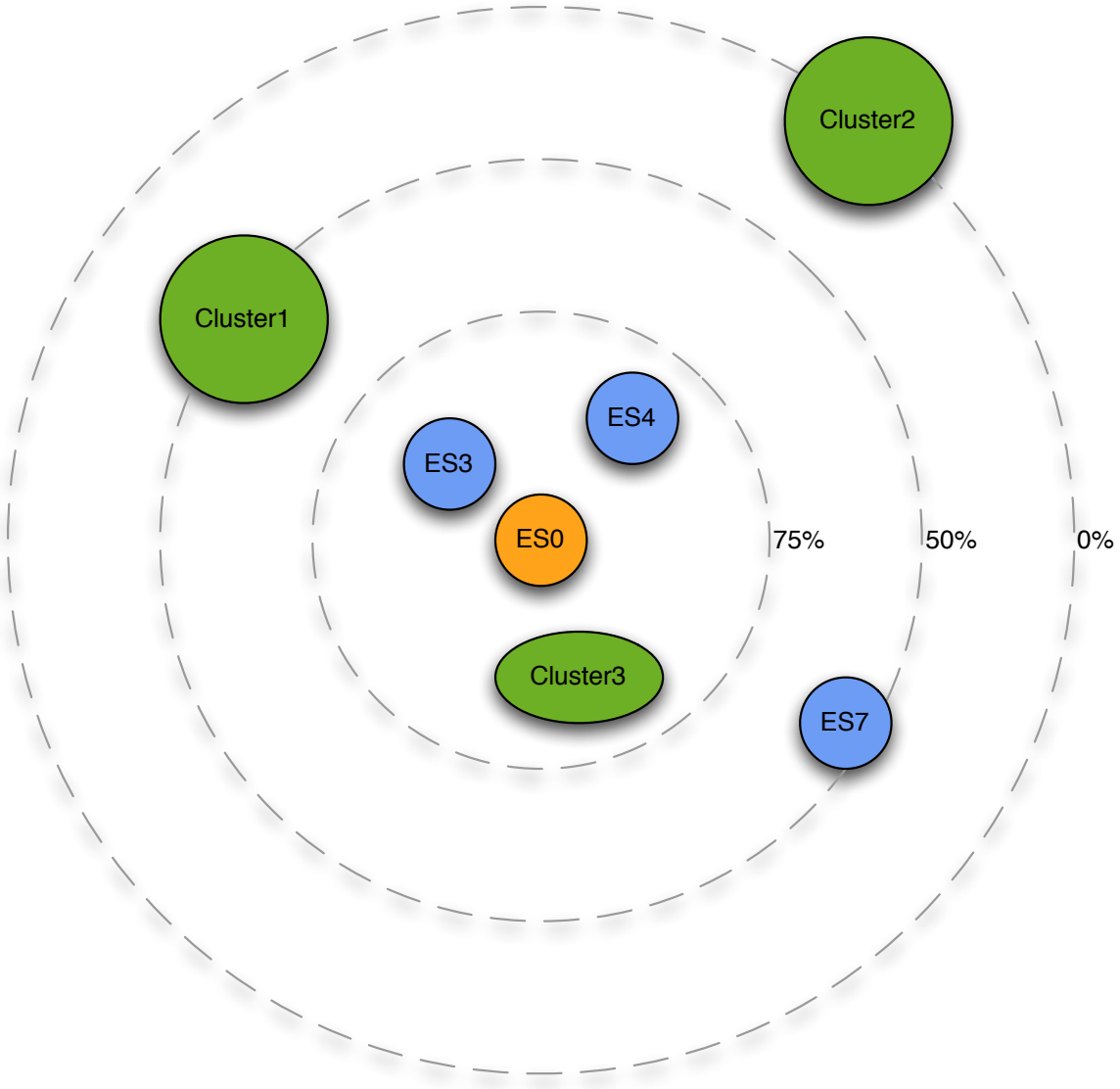

Figure 4.19: The determined similarity of the overall event streams can be used to display and cluster them in a radial chart. In this example event stream 0 (ES0) serves as the origin of the plot. Event streams 3 and 4 are very similar to ES0 and cluster 3 is also similar to ES0. The three are, however, not very similar to one another or they would be drawn closer together. The same is true for clusters 1 and 2 as well as event stream 7 which are all quite unlike event stream 0, but also very unlike one another.

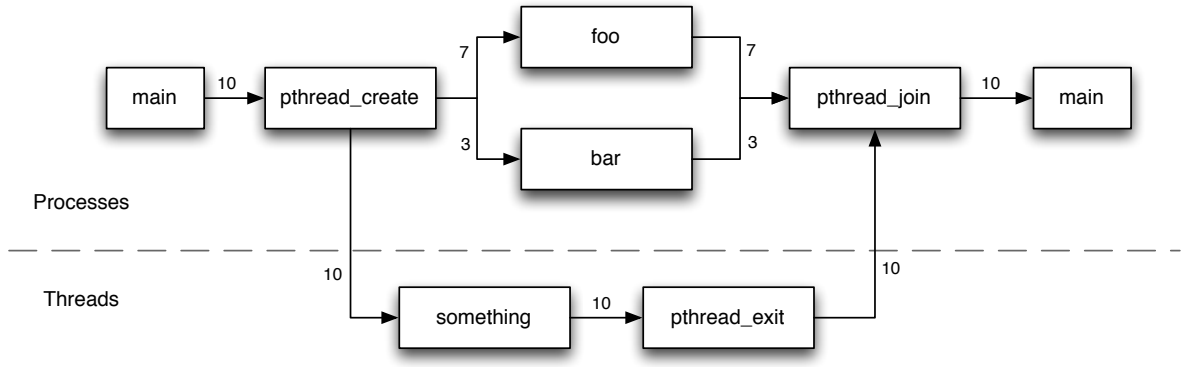

Figure 4.20: The PPFG can show the thread activity in a separate part of the display area. The parent state of a launched thread then has two following states.

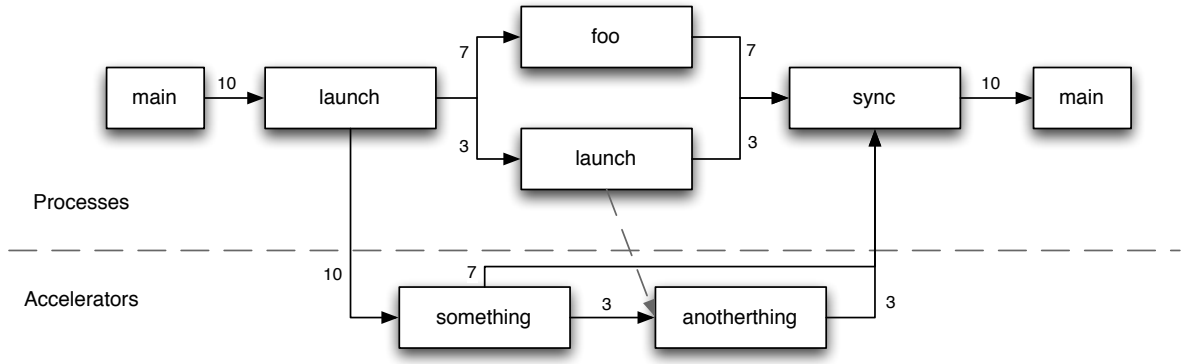

Figure 4.21: The PPFG can show the accelerator activity in a separate part of the display area. The host process then launches work on the accelerator. The synchronization operation fuses the divergent program flows. A dashed line connects asynchronous kernel launches between the host and the device.

5 Studying Multi-Hybrid Application Performance

After presenting multiple approaches for studying the performance of applications using hardware accelerators in the previous chapter, they will be put to test using a highly scalable, multi-hybrid, real-life application. The introduced adaption of the Particle-in-Cell algorithm is a joint project of the groups of Dr. Bussmann at the Helmholtz-Center Dresden-Rossendorf (HZDR) and Prof. Dr. Nagel at the Center for Information Services and High Performance Computing (ZIH) at the Technische Universität Dresden. It has gained attention in the plasma physics community for currently being one of the fastest fully relativistic simulation of its kind. An important part of this success story was the availability of a performance tool that enabled the detailed identification of performance bottlenecks as well as studies on the effect of various performance enhancement strategies in the code. This chapter introduces the simulated problem with its computational science implications, presents how the developed methods from Chapter 4 contributed to enhancing the application, describes the current status and planned future work on the application, and evaluates the developed methodology when applied to the application.

5.1 Particle-in-Cell Simulations

Plasma physics relies heavily on computer simulation to understand the complex inner workings of this technically almost unobservable state of matter. The plasma state can be described by kinetic or fluid simulations or a combination of both. This section is a brief introduction into Particle-in-Cell simulations and based on [BL95]. Kinetic simulations consider the plasma as an ensemble of charged particles interacting in the presence of an external electromagnetic field. This can either be done numerically by solving the underlying kinetic equations discretizing for example Vlasov or Fokker-Planck equations or by so called "particle" methods which aggregate a number of particles into a single ensemble described by a charge distribution function —also called macro-particles—and compute the motion of these macro-particles due to the electric and magnetic fields. Fluid simulations on the other hand treat the plasma as a fluid and determine how the matter in the plasma is transported using magnetohydrodynamic (MHD) equations. In this case an ensemble average is applied to the plasma. A combination of both general approaches is also possible, so that some parts of the plasma are simulated using a particle method while the rest is simulated using an MHD method.

The Particle-in-Cell (PIC) method is an implementation of the above mentioned "particle" method. PIC can use a particle-particle (PP), a particle-mesh (PM), or a combined (P^3M) scheme. In a particle-particle scheme the simulation only covers short-range interaction of the particles, in particle-mesh schemes only via long-range forces mediated over a mesh is considered. P^3M schemes combine the latter two approaches and compute both short-range and long-range forces. The mesh is used to interpolate the electric and magnetic field; the simulated particles are macro-particles with a mass m and a charge q using a particle density distribution function. The components of the mesh are the so called "cells" that

provide the name for this approach. The computational difficulty in this simulation method is that the simulated particles can move freely—albeit governed by the laws of physics—in the simulated area. Hence, finding a well performing mapping between cell and particle data is one major consideration. Another challenge is that very large simulation areas require a domain decomposition of the mesh and, thus, also of the particle data. This decomposition not only results in the typical communication of border halos to the corresponding neighbors for the mesh data, but also in particles leaving and entering domains.

The PIC method simulates four different steps for all particles and all cells for every simulated time step (also shown in Figure 5.1).

1. Compute the Lorentz force acting on every macro-particle using the current distribution of the electric and magnetic field stored in the mesh.
2. Integrate the equation of motion for all macro-particles and determine a new position and velocity for them.
3. Moving electrical charges give a current which is determined from the new positions and momenta of the macro-particles.
4. Use Maxwell's equations to update the electric and magnetic field from the currents and start over.

In an implementation of the PIC method the first two steps are typically combined into a so called "particle pusher". The particle pusher iterates over all particles, determines the Lorentz force on them and updates their position based on how far they will travel due to this force in the simulated time step. If collisions or particle-particle interaction is ignored in this step, all particles can be updated in parallel. Collisions are usually just approximated using a Monte-Carlo methods since an exact simulation of the collisions would be to time consuming and is not really necessary, especially in the case of strong external fields.

The simulated area can be either two- or three-dimensional, the algorithm is then characterized as 2D3V or 3D3V where the "3V" means that all velocity vectors, electric and magnetic field vectors are three-dimensional. In the 2D3V variant position vectors are only two-dimensional and position changes in the third dimension are ignored, while in the 3D3V case position vetcors are three-dimensional. The data structure used for the mesh data is thus a two- or three-dimensional matrix for electric fields, magnetic fields and currents. Finding a well performing data structure for the particle data is more complicated. The computational steps of PIC require a close interaction between mesh and particle data, therefore, particles that are close together should also be stored closed to one another from a memory location point of view. This spatial data locality provides a good data cache hit ratio, which is essential for good performance on a modern cache based microprocessor. One possibility to implement a particle data structure is a linked list of particles [Bow01, BAB+08]. The linked list can be resorted from time to time in order to preserve the required data locality between particle and mesh data as shown in Figure 5.2.

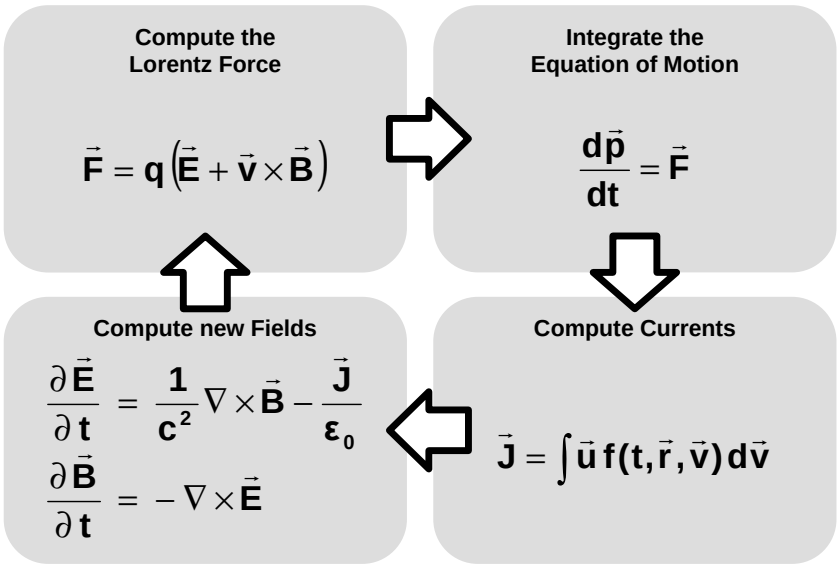

Figure 5.1: The particle in cell algorithm is processed in four steps for every time step it simulates: First the Lorentz force on all charged particles is determined, then their movement according with these forces computed. The moving charges induce a current which in turn changes the electric and the magnetic field. This will also change the Lorentz force and one starts all over again. [Bus12].

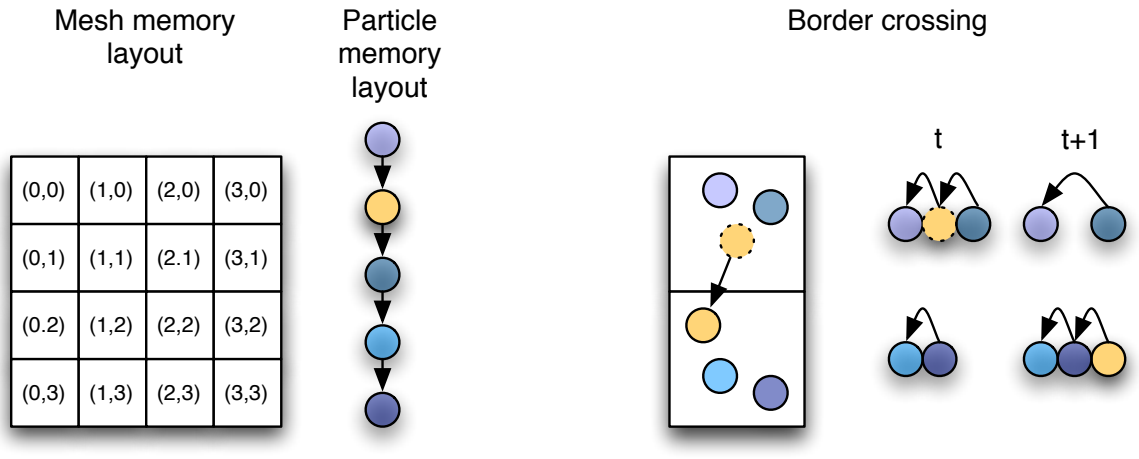

Figure 5.2: One standard approach to store the dynamic particle data is the usage of a linked list. The particles need to be stored in the order of the cells they are in. In this example the particles from the upper cell need to be stored in the list prior to particles from the lower cell. When the computation steps now iterate over particles and cells, they will experience a high data locality. In order to preserve this locality the list has to be resorted when particles cross the border into another cell. [BWH+10]

5.2 Towards a Hardware-Accelerated PIC-Solution

The computational steps of the PIC method expose a very large amount of parallelism. As mentioned before all particles and mesh points can be updated independently. The massively parallel nature of hardware accelerators seems to be a good fit and promises a significant speedup in the simulation. The cooperation between HZDR and ZIH started after BURAU (HZDR) developed a CUDA accelerated single GPU version—called PIConGPU—of a very simple PIC method and delivered a significant speedup compared to a standard CPU version. This particle-mesh PIC implementation simulates how a very high energy laser pulse accelerates charged particles to relativistic velocities. In this case particle-particle interaction can be neglected since the long range forces induced by the electromagnetic field of the laser pulse exceed the close range forces between the particles by several orders of magnitude. A hybrid approach combining the PIC method with a fluid simulation was not yet considered since it requires a large amount of particles to generate ensemble-averaged temperatures from the particles, which was up until now limited by the total memory available on GPU clusters. [Bus12]

The limited amount of memory on one GPU also limits the simulation size of a single GPU PIC simulation. Hence, a multi-hybrid version that preserves the already established speedup is highly desirable. In order to support this project, ZIH contributed its experience in MPI and an thread parallel applications as well as the performance tool extensions developed as part of this thesis. An overarching goal of this joint effort was to develope a library that encapsulates the GPU device handling, specific mesh or particle data structure implementations, as well as the inter-GPU communication. Furthermore, the developed library should have the flexibility to include other PIC variants that for example cover particle-particle interaction or the fluid dynamics of the simulated plasma. It hsa been decided that all computational work should be done on the GPU, while the CPU is reserved for supporting work such as communication and file I/O. Hence, one objective of the project is to find ways to always keep the GPU busy with work.

Introducing and Optimizing MPI-Communication

The decomposition over multiple GPUs and multiple hosts follows the same scheme that distributes a PIC problem using just MPI: the mesh is split into as many parts as GPUs are available, padded with a communication halo, and each GPU receives its submesh. Furthermore, all particles that are positioned in the submesh are also placed on the GPU. The communication principle is also the same as with an MPI accelerated PIC implementation: the mesh data for the border cells is exchanged with the corresponding neighbor and particles leaving one submesh have to be transferred to the correct neighboring GPU. However, one difference is that a direct communication from one GPU to another using MPI is not yet possible. As a result a data transfer between two GPUs currently involves a data transfer from the source device to its host, an MPI message from the source host to the target host, and a data transfer from the target host to the target device.

The performance analysis of a straightforward implementation—as shown in Figure 5.3—of this approach reveals a very low GPU device utilization due to high communication latency. In order to overcome this undesired effect, two steps have been taken. First, the used GPU cluster has been equipped with an Infiniband network to reduce the rather high hardware limit for communication latency in the previously used Gigabit Ethernet network. Second, computation on the GPU and communication by the

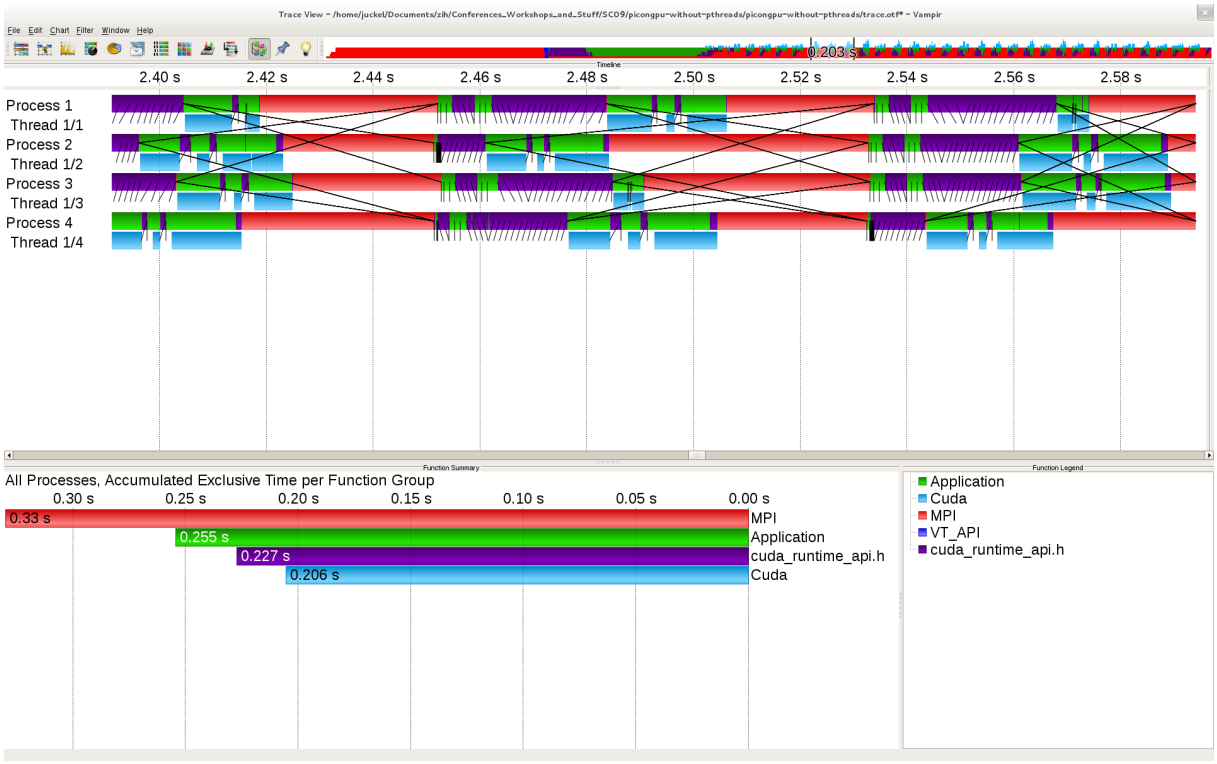

Figure 5.3: The Vampir Master Timeline and Function Summary of a straightforward MPI parallelization of PIConGPU show that the MPI communication is dominating the overall program execution. The actual compute part—the blue CUDA parts—is of the shortest duration.

CPU needs to be interleaved. Since a typical computation pattern in a PIC method is the Yee lattice as shown in Figure 5.4, one computational step on the mesh data can be split into two parts: a core and a border. The core region has no dependency on data from neighboring GPUs and can be computed using data already present on the device. During this computation the data transfer from the neighboring GPUs can take place and the computation of the border region is delayed until this data has arrived. If the core region is significantly larger than the border, which is typically the case, the runtime of the computation on the core regions exceeds the communication latency and the GPU can be kept busy during the data transfer.

Another contributing factor to the poor GPU utilization is that the host process is responsible for both the device management and MPI communication. The long `MPI_Waitall` calls block any other parallel activity on the host. As a first solution the MPI communication was moved to an additional host thread which enables the parent host process to be available for the GPU to supply new data or work. The result of these steps is a much improved GPU utilization , as shown in Figure 5.5.

Optimizing Time Consuming PIC Steps

After making sure that the GPU is always occupied with work, the the question of the quality of this occupation or how well the GPU kernels actually utilize the device capabilities remains. The analysis of the GPU activity—as shown in Figure 5.6—lists the most time consuming kernel as `cptCurrent`. Most interestingly, the runtime of all kernels also increases with increasing overall runtime of the application

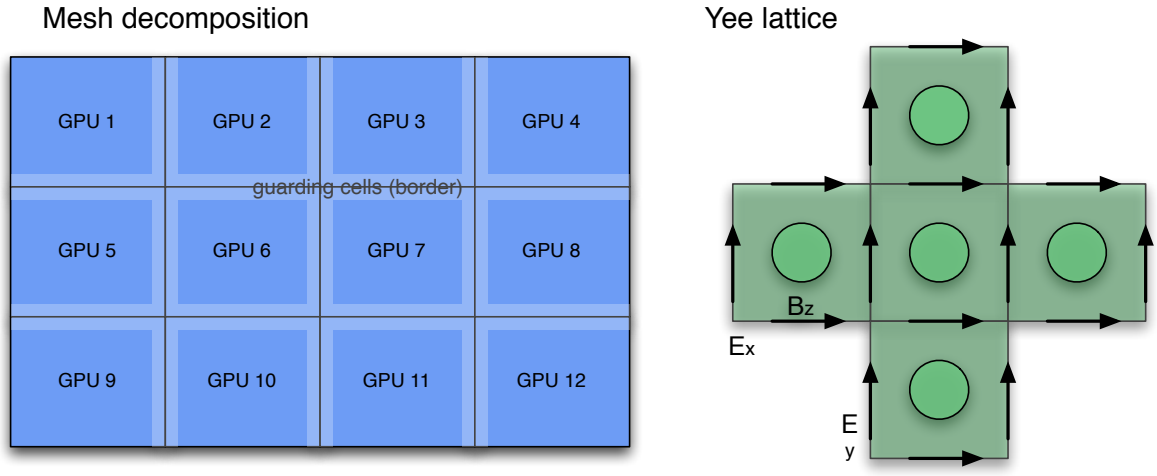

Figure 5.4: PIConGPU uses a standard domain decomposition approach for the mesh data (left). Each subdomain uses a one cell wide border which is used for communication with its neighbor. The computational scheme in the mesh data is an incomplete stencil operation (Yee lattice scheme). This increases the core area which can be computed without any input data from the neighbor even further. [BWH+10]

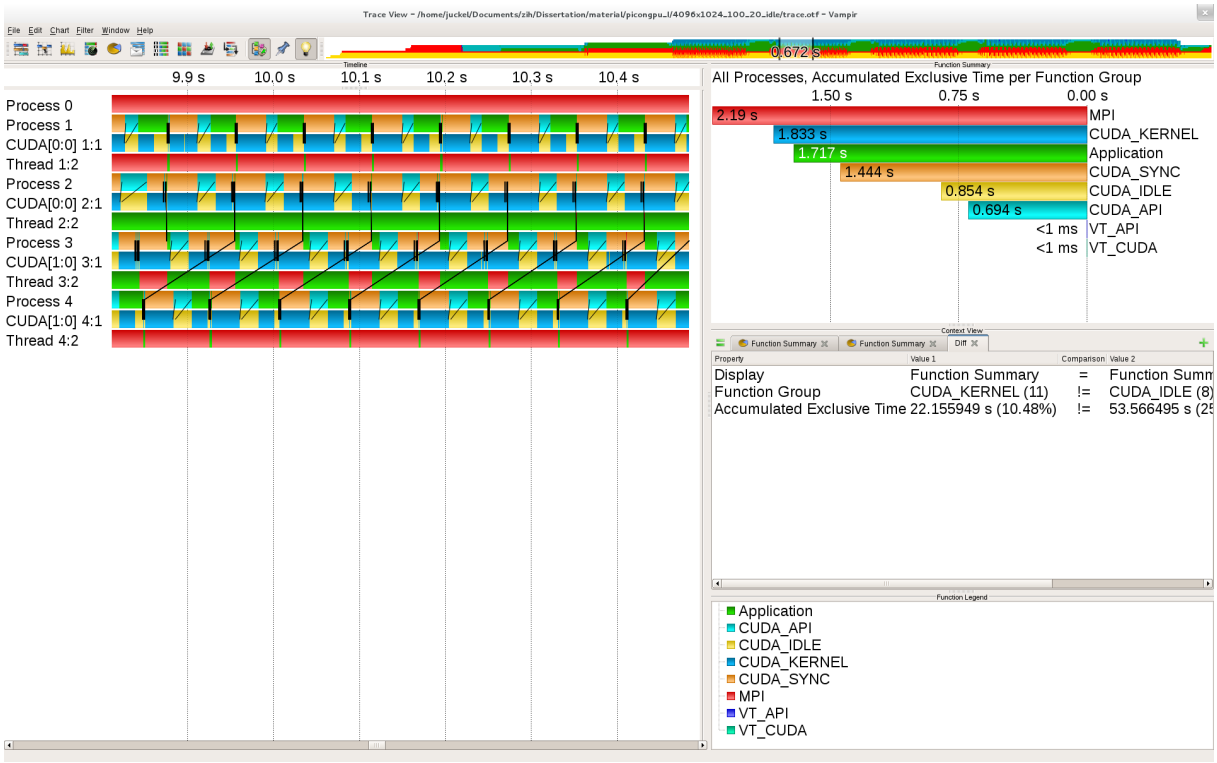

Figure 5.5: The Vampir Master Timeline and Function Summary of a combined MPI and pthread parellization of PIConGPU shows a much increased CUDA utilization compared to Figure 5.3. MPI communication is still a large part of the execution time of the overall application, but only the send operations are carried out by the host process. The receive operations are offloaded into a host thread so that the host process can launch more work on the GPU.

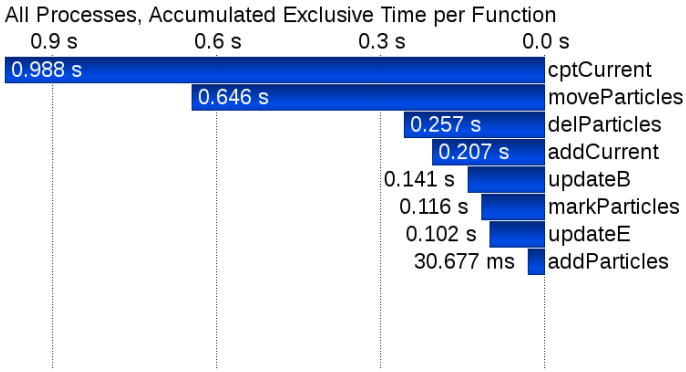

Figure 5.6: The runtime profile for all CUDA kernels of one iteration of PIConGPU reveals that the computation of the current and the movement of the particles are the most time consuming operations. Hence, they should be the target of a kernel optimization.

as shown in Figure 5.7. This increase in runtime is due to the fact that the motion of the particles leads to an increasing amount of non-coalesced memory accesses on the GPU and, thus, to a longer runtime for one iteration of the code. The poor memory utilization originates from the usage of a linked list. As particles travel, the list needs to be searched until all particles of a cell are found when the kernel updates the mesh data. Vice versa, a kernel iterating over the particles access mesh data from potentially very distant cells.

The proposed solution to this is the introduction of a new data structure for the particle data and a better localization of both particle and mesh data [HSW+10]. First of all the mesh data needs to be aligned on superword boundaries so that the GPU can access the data in a coalesced manner. Furthermore, the mapping between a thread block on the device and the mesh data is formalized by introducing supercells. A supercell is a subregion of the simulation area of one GPU where each cell of the subregion is mapped onto one thread of the thread block for kernels that update the mesh data. The particle data is vectorized by placing it in so called tiles. A tile is an assembly of fixed size arrays with one array for each property of the particles. One of these properties—as shown in Figure 5.8—is always a cell index, so that a fast mapping between cells and particles is possible. The array size is the same as the number of threads in the thread block, thus, allowing a mapping of one thread per particle. If a supercell contains more particles than one tile can hold, an additional tile will be added and linked to the first tile. One thread is then responsible for two particles, one from each tile. More tiles can be added in the same manner. When a particle leaves as supercell, the cell index at its position is set to -1 to mark the position as free. An entering particle is placed onto the first free position or, in case none are available, at the end of the tile. The advantage of this high locality data storage approach is that kernels that read mesh or particle data multiple times can benefit from the data being copied to shared memory and there being accessible with zero latency. As shown in Figure 5.7, the introduction of this new data structure has not only improved the overall runtime for one iteration of the code, but also provides an almost constant runtime over the whole program execution.

Overloading the GPU Device

The improved data structures and kernels preserve a good GPU utilization—as shown in Figure 5.9—while reducing the overall computation time. However, the GPU utilization by the host process itself

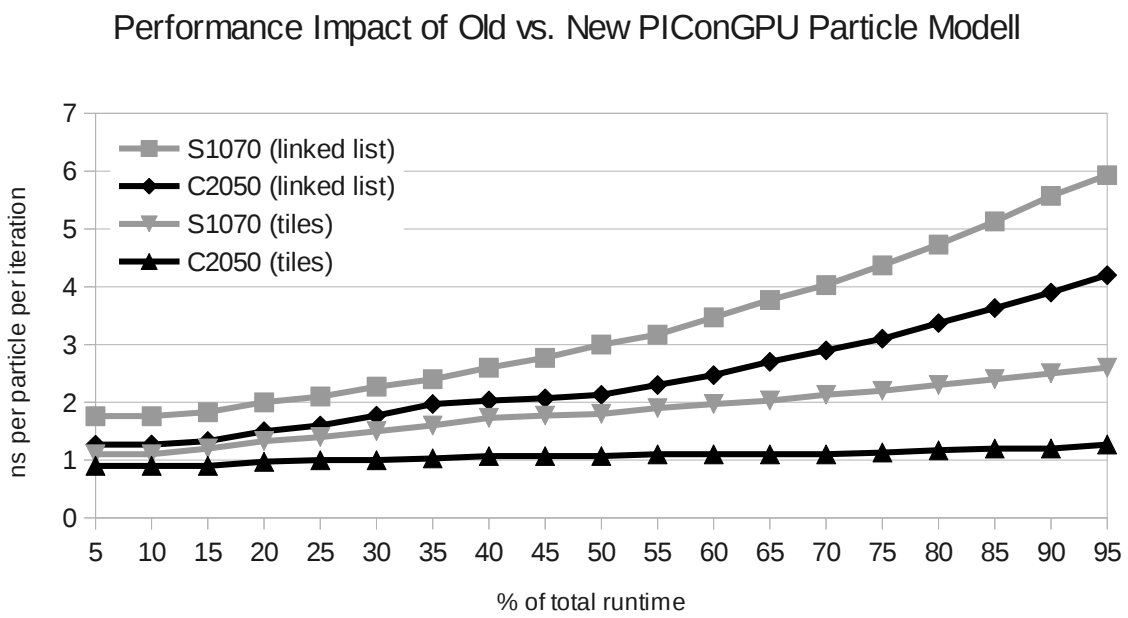

Figure 5.7: The performance impact of using the new tiles instead of a linked list for the particle data is clearly visible. While the time to complete on iteration is constantly increasing when using a linked list, the effect has almost disappeared for the tile data structure on the Fermi GPU (C2050). [HSW+10]

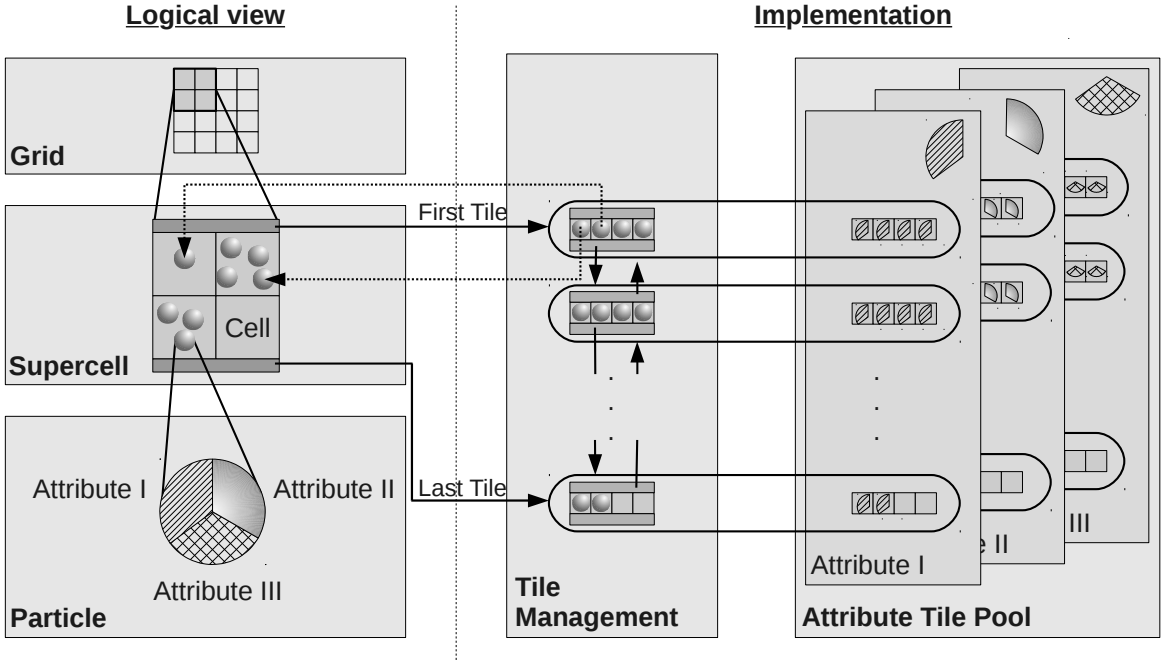

Figure 5.8: The whole simulation mesh can be subdivided into supercells that correspond to one thread block on the GPU. The particle data is stored in multiple short vectors called tiles. The size of the vectors is also equal to the thread block, hence a GPU thread can be mapped to a cell as well as a particle in the supercell. The first tile always references the cell index where the particle is currently positioned, the other tiles store other attributes of the particle such as the exact position and the velocity. The new mapping in combination with the usage of shared memory makes sorting of particles within the supercell obsolete. [Wid12]

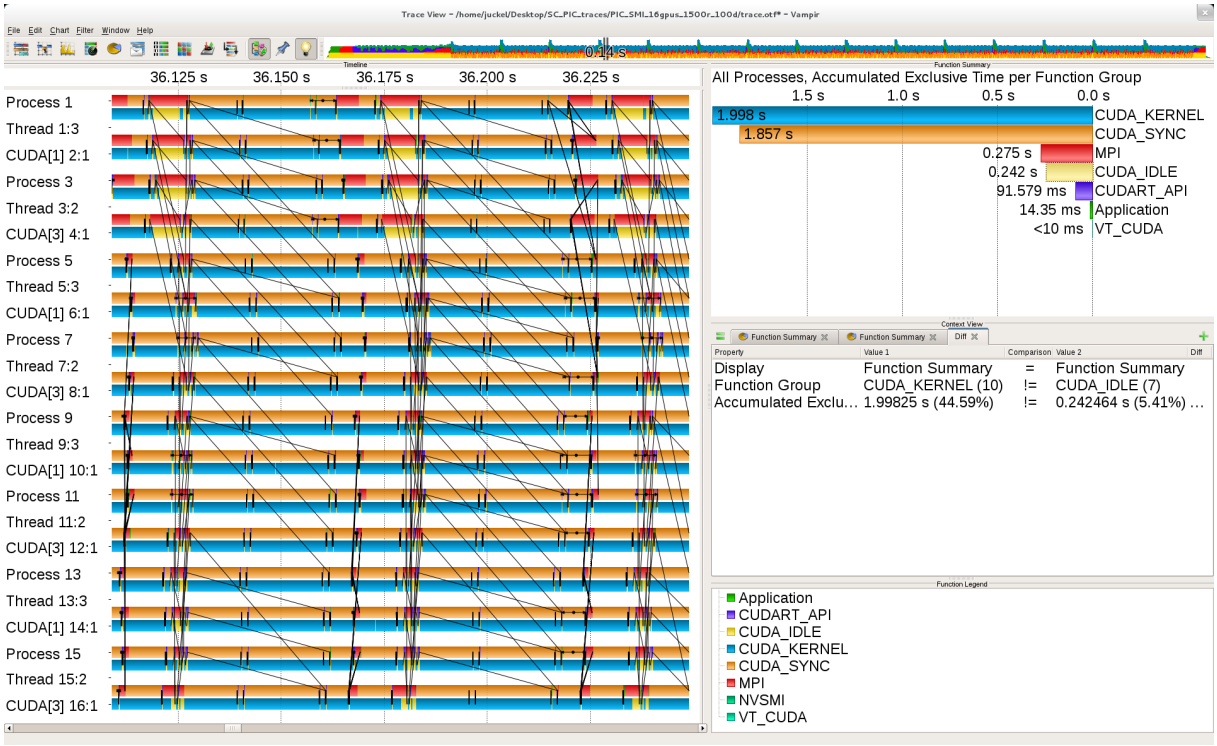

Figure 5.9: The Vampir Master Timeline and Function Summary of a PIConGPU version after kernel optimization shows a much improved device utilization. The MPI behavior has also changed in this version to the usage of asynchronous communication, so that no additional communication thread is needed any longer.

still is one sequential execution stream. At the same time the computational steps in the developed PIC implementation also exhibit a level of concurrency. One example is that the computation of a border region of the electric field $\vec{E}$ can be run in parallel with the computation of the magnetic field $\vec{B}$ for the core region. This is especially helpful, since the border region is not large enough to run enough threads to utilize the full GPU. Furthermore, data transfers can also be overlapped with computation.

PIConGPU supports the concurrency by introducing its own event system that is mapped onto GPU events. The concurrent execution of kernels on the GPU is managed by using multiple execution streams. Every stream can be controlled by device events, which hold the execution of kernels from that stream until an subscribed event has occurred. The programmer can specify dependencies of one kernel to another and the underlying library will automatically place the kernels in the correct streams and insert events where necessary.

5.3 Current Status and Future Work

With all these improvements PIConGPU demonstrates a quite impressive scalability on one of the largest GPU systems in the world as shown in Figure 5.10.

PIConGPU can also demonstrate its capabilities in a different manner. The application has been accelerated to the point where even large scale runs can immediately display the simulation data. With about two to three frames per second this visualization and simulation is unique in its speed and resolution.

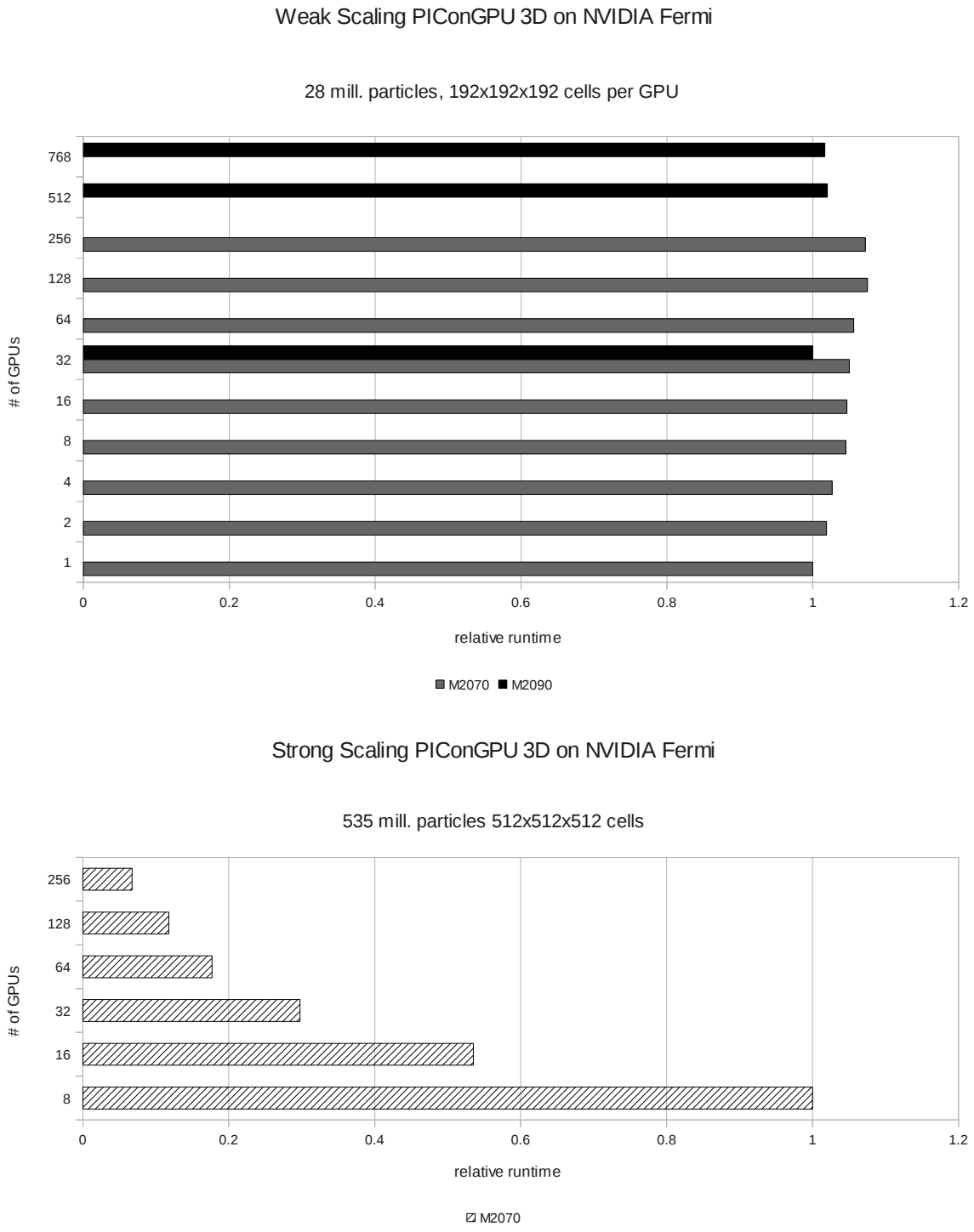

Figure 5.10: Large scale tests with PIConGPU show a very good weak scaling. The largest test was run on 768 NVIDIA Fermi GPUs. The strong scaling is naturally limited, but still shows a good efficiency. [Wid12]

Future plans include the adaption to a virtual environment, so that physicists can modify the simulation as it progresses, restart it with changed parameters, and, in general, use it to discover new effects or better laser targets.

The main achievement of the PIConGPU project is, however, the separation of the physics from the rest of the application. This rest was encapsulated into a library called “libgpugrid”, which offers all the generic routines needed to program similar simulations that use different physics algorithms. This library offers the data structures for field data like electric and magnetic fields that will be mapped onto the mesh cells and supercells but also the high performing particle data structure. The data is associated with a simulation region, which can also have neighbors. This neighbor relationship is used to carry out all communication as soon as the library is told that results are ready for communication. The event system is also part of the library. As a result the physicist can use high level templated data types and their corresponding methods to treat the parallel program as if he is programming just one GPU.

Due to the scaling of the current PIConGPU implementation and the flexibility of the underlying simulation library new simulation parts could be added to the overall simulation. When charged particles change their movement they emit an electromagnetic radiation. In order to determine this radiation, one is currently limited by the cell size of the PIC simulation. The shortest wavelengths that can be determined are the length of one cell. Since PIConGPU’s particle pusher determines the exact trajectories of all moving particles, this information can be used to determine the radiation and, thus, greatly increase the resolution of the determined wavelengths. This simulation is so computationally intense that it was previously impossible to run on the full simulation area. Other methods that are to be included into PIConGPU and the underlying libgpugrid library in the future are particle-particle interactions and fluid simulations of the plasma. The main issue with MHD methods is their need for a lot of particles per cell in order to generate a temperature distribution from the particle data. This is only possible in some parts of the simulation area. Hence, a hybrid model using a particle method in most areas and using a MHD methos in some areas will be implemented.

The analysis of the event log of the latest PIConGPU version still reveals on major issue: load imbalance. Since the simulation area is divided equally among all GPUs, during the simulation some GPUs might end up with more particles than others. The time needed for one computing step is directly depending on the number of simulated particles. Hence a distribution of the simulation area among the GPUs so that the number of particles per GPU is constant would provide a much better balancing. Such a distribution, on the other hand, would require the mesh to be adapted quite frequently and would also increase the communication needs of the application due to the redistributed mesh data. Albeit these difficulties, this mesh adaption is the logical next step to increase the overall performance of the application even further and is under development as well.

5.4 Application of the Developed Methodology

The developed performance measurement techniques could demonstrate their scalability and usefulness with PIConGPU as an example. The Vampir visualization of a PIConGPU run using 512 NVIDIA Tesla M2090 GPUs is shown in Figure 5.11. In this case the measurement overhead is 29% and the size of the generated event log is 37 GB. The GPU utilization as shown in Figure 5.12 is at 71%. While this

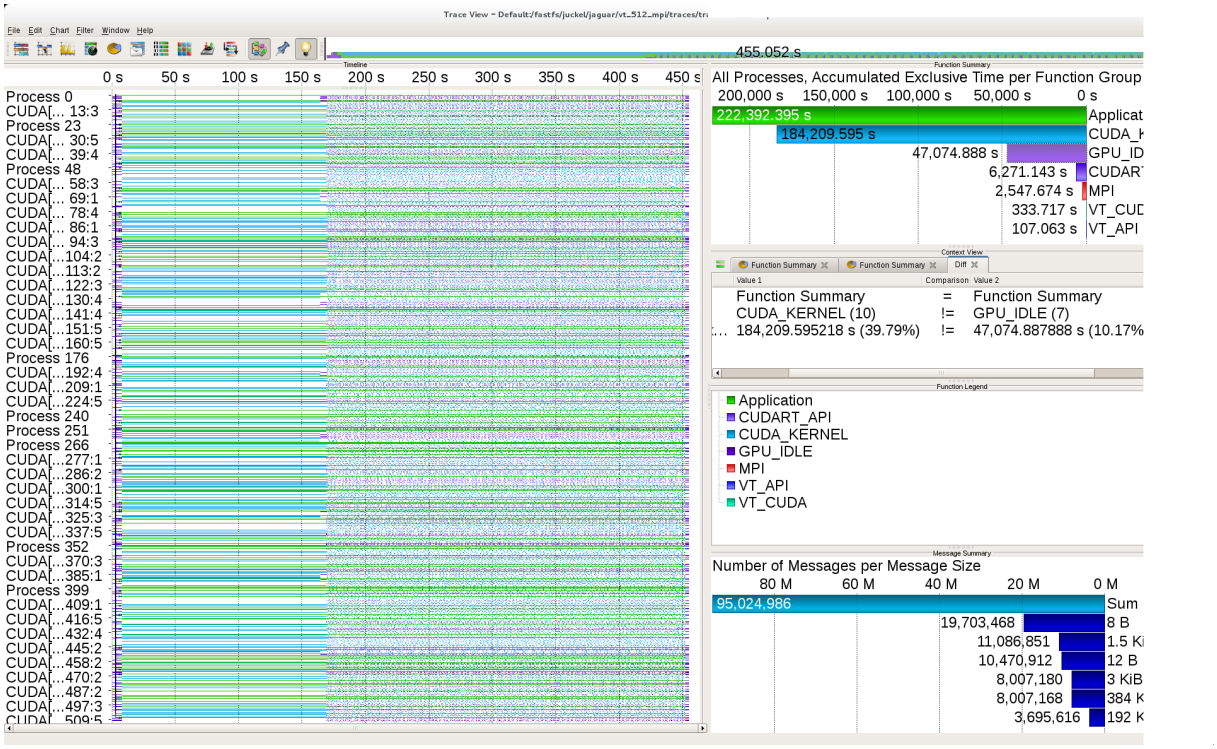

Figure 5.11: A Vampir screenshot for the 512 GPU run of PIConGPU shows the already presented issue with the timeline and message profile aliasing.

is a lower utilization than before, the overall iteration time is still lower than in earlier versions. The optimizations on the code have reached the point where the execution time follows the critical data path as shown in Figure 5.13. The GPU_IDLE times are used for data transfers between the host and device, computation can only resume once required data has arrived. The last time line display also reveals a weakness of the used NVIDIA Fermi architecture: while it is supposed to support concurrent stream execution, the trace file shows that the streams are in fact serialized. The problem was determined to be a poor device implementation of events that are used by PIConGPU to synchronize the parallel streams. The on-device events suffer from the same limitation as the timing events, they serialize the GPU execution.

The evolution of the PIConGPU simulation and the analysis methods presented in this thesis went side by side. Event logs of the early version of PIConGPU (Figures 5.3, 5.5, and 5.9) are recorded with VampirTrace using the event-based measurement method from Section 4.1.3. The evolution of the recording can even be seen in the screenshots: in the early versions of the CUDA event logging (Figure 5.3), the GPU event streams were labeled as simple threads using the thread mechanism of VampirTrace. In the later versions this has been replaced by a “CUDA” label that includes the device ID and the stream ID of the recorded event stream as well. The label “CUDA[0:8] 3:4” in Figure 5.13 for example means that this event stream is the forth child of Process 3 and a CUDA event stream of device 0 stream 8.

The large scale traces have been recorded with a VampirTrace version that uses the CUDA Profiling Tool Interface (CUPTI) as described in Section 4.2.3. A comparison of execution time, event log size and overhead for recording various levels of parallelism is shown in Table 5.1. The execution time recorded is only the time for the simulation without initialization and cleanup of the code, which would also

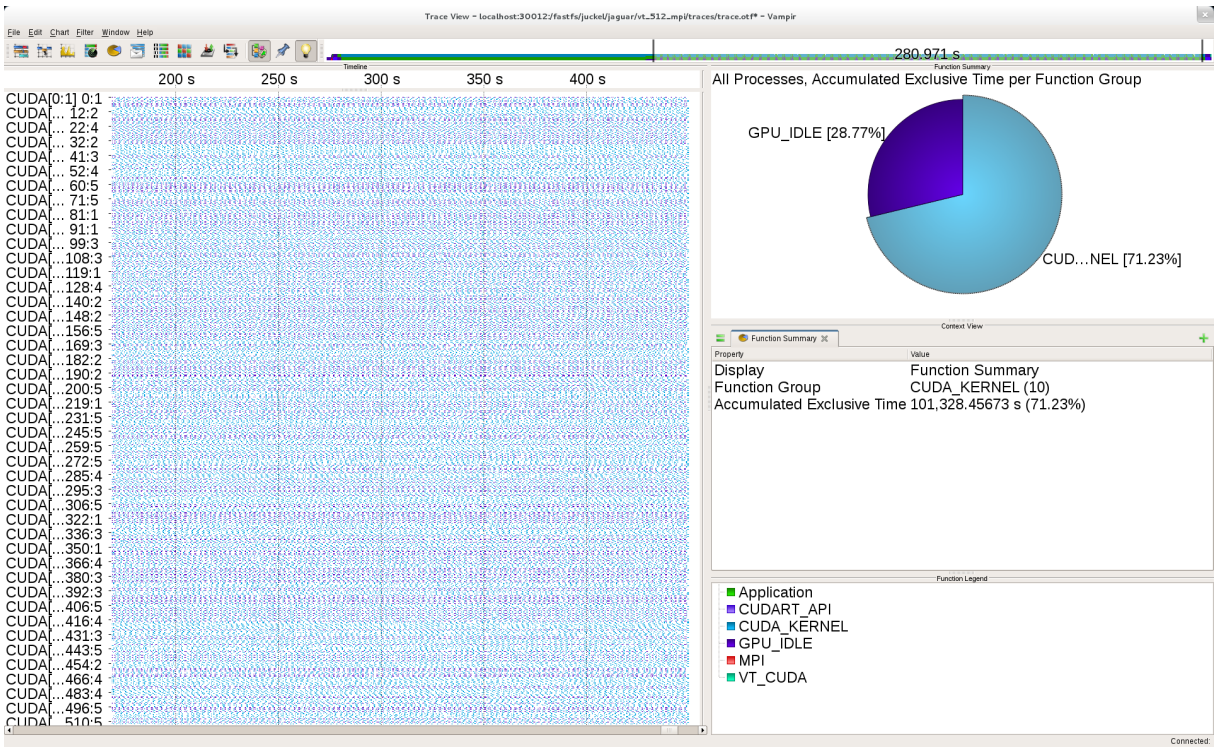

Figure 5.12: When displaying only the GPU streams of the same 512 GPU trace, the timeline is still distorted by the unequal utilization of the device via the various streams. Nevertheless, the filtered event log can now be used to profile the CUDA kernels. In this case the overall GPU utilization is 71.23%.

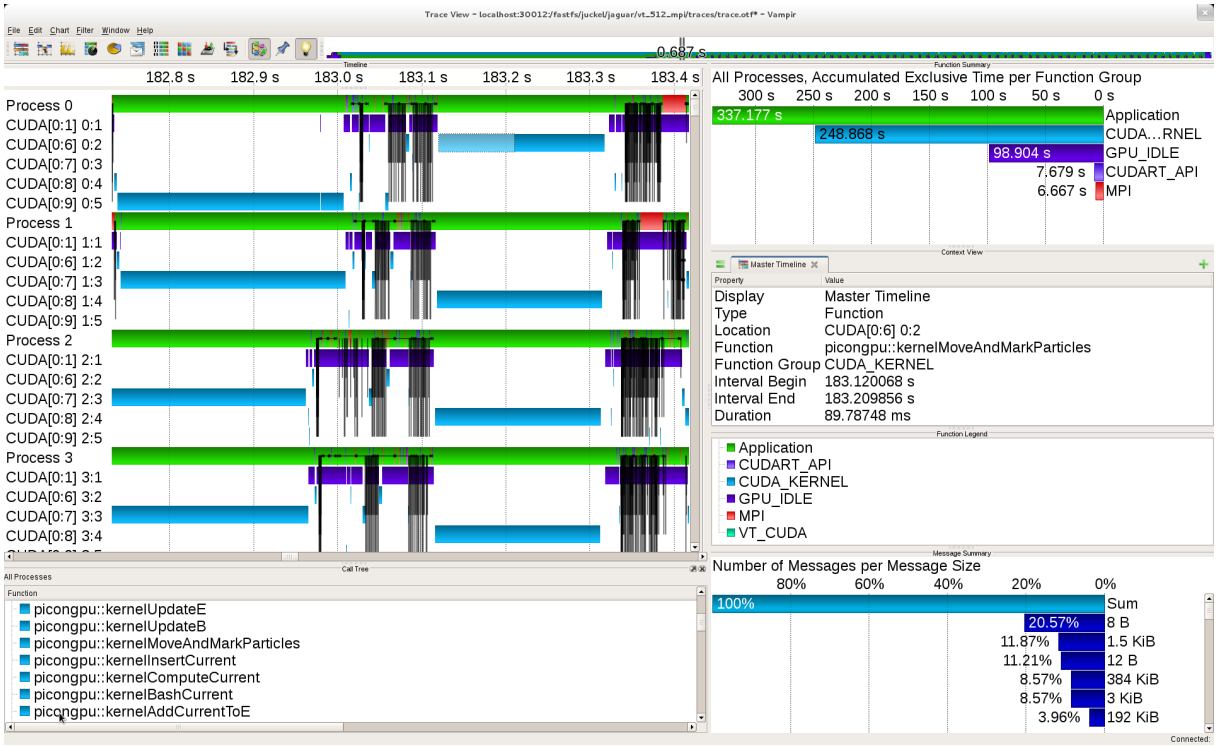

Figure 5.13: Vampir can also show one iteration of PIConGPU in all detail. In this case the host-device interaction is shown. The message profile shows the number of host-to-device transfers listed by transfer size.

Table 5.1: Comparison of three different PIConGPU runs (running for 400 iterations) on the Titandev system in Oak Ridge National Laboratory using 512 NVIDIA Tesla M2090 GPUs.

run type	execution time	event log size	overhead
no instrumentation	222 s	-	-
only host logged (no MPI, no CUDA)	261 s	2.4 GB	18%
host and CUDA logged (no MPI)	**268 s**	**7.3 GB**	**21%**
host and CUDA logged (MPI filtered)	285 s	7.3 GB	28%
host, CUDA, and MPI logged	287 s	37 GB	29%

Table 5.2: Comparison of the number of events of the event logs from Table 5.1

run type	host event count	GPU event count
only host logged (no MPI, no CUDA)	465,545,851	-
host and CUDA logged (no MPI)	968,358,572	352,763,161
CUDA logged (MPI filtered)	968,308,495	352,760,588
host, CUDA, and MPI logged	7,569,076,954	352,762,630

include the recording of the event log. It can be seen that the overall increase in the execution time is below 30% even when logging all activity. The largest portion of this overhead is due to monitoring the host activity. Table 5.2 shows the number of logged events for the different event logs. The over 400 million host events in the first case cause a 39 s increase in the execution time of the application, a lot of which is due to the hundreds of billions events that are filtered. The application is a highly modular C++ program and only the methods driving the simulation have been recorded. Nevertheless, all events are monitored first to decide whether to record them or not, which results in the measured overhead.

The 3% overhead for monitoring and recording the CUDA activity on the host and device is rather low, compared to the host and MPI monitoring. The CUDA activity adds events to the GPU and the host event streams, because the CUDA runtime library usage by the host is recorded as well. When the inter-node communication with MPI is also monitored and recorded, the runtime increases another 8%. Similar to the host monitoring this overhead is mainly due to the monitoring itself, which causes most of the execution time increase. The additional 6.5 billion MPI events are also a filtered set of all MPI events, since the application heavily uses `MPI_Test` for the asynchronous communication. In this case only the last `MPI_Test` for a communication—which corresponds to the receive event—was recorded in the event log.

The development of PIConGPU benefited greatly from being accompanied by a very flexible and capable performance tool. With the detailed monitoring capabilities of VampirTrace and its CUDA extension, bottlenecks in the parallelization could be detected very early in the development process. The GPU monitoring using the CUPTI environment offers a very precise, low-overhead performance monitor that extends the capabilities of VampirTrace to also cover modern hybrid supercomputer architectures.

6 Conclusion and Future Work

The thesis develops generic methods to extend performance analysis techniques for parallel applications to hardware accelerators. When hardware accelerators arrived as a novel form of computing devices, performance tools were limited to a one host with one device setup. A variety of different hardware accelerators is introduced at the beginning of this thesis. Commonalties between the different hardware designs are established and programming models are explained. Event logging is a special form of application performance analysis that provides the greatest level of detail but also has the largest data volume. The thesis presents the state-of-the-art in performance data recording and visualization and provides an overview of performance tools for hardware accelerators.

This thesis is dedicated to developing novel methods for extending event log generation and visualization for performance data gathered from hardware accelerators. The thesis discusses generic possibilities for acquiring performance data on three different levels: just by observations on the host, by providing a special driver interface, and by directly generating the data on the device. The additional layer in parallelization results in ambiguous performance displays when using existing visualization tools. An approach to filter relevant information when using traditional visualization techniques and a novel visualization approach using performance flow graphs is developed. The new presentation approach provides a comprehensive view of the event log and enables a direct navigation to parts of the application with structural or temporal differences. This thesis demonstrates the capabilities of the developed methods in the study and enhancement of a multi-hybrid real-world application.

The novel contributions are a hardware accelerator specific performance tool interface proposal and its implementation with NVIDIA CUPTI, and the parallel program flow graphs for a more concise presentation of the performance data. As a foundation it utilizes a generic approach for recording an applications interaction with any kind of API using an automated library call interception. An additional success of the presented work is the recognition by Oak Ridge National Laboratory with a special contract to include the work in the Vampir and VampirTrace products. The new HPC system in Oak Ridge, called “Titan”, will be a hybrid system featuring 18,688 NVIDIA Kepler K20 GPUs alongside 299,008 standard AMD Opteron processor cores. Aided by the extensions to VampirTrace, that were implemented using the methods presented in this thesis, a complete system view of an application using all available becomes possible.

While the thesis is successful in providing a platform independent approach for including hardware accelerator performance data, a full picture including all possible data can only be obtained with a novel interface as proposed or a hardware specific solution. Hence, future work must focus on hardware accelerator programming models to include a performance measurement interface as well. Especially directive based programming models require both a high level mapping of performance data to the introduced directives but also the low level data from the underlying hardware. A joint development of the Center for Information Services and HPC (ZIH) with CAPS Enterprise developed such an interface between HMPP and VampirTrace.

As a consequence of this thesis, ZIH has joined both the OpenMP and OpenACC forum, since the most efficient performance analysis is only possible when interfacing directly with the accelerator driver. OpenMP 4.0 will be able to offload work onto hardware accelerators as well as host processor threads. The OpenMP 4.0 workgroup has recognized that a tool interface is vital to the success of a new or enhanced programming model. Programmers will then be able to see how the compiler offloaded directives to the accelerator and understand how well their program utilizes the hardware.

Upcoming new hardware like the NVIDIA Kepler GPUs or the Intel Xeon Phi (MIC) processors will put the developed methods to the test. Initial information on the new hardware underlines the importance of a common performance tool interface. For the NVIDIA Kepler GPU this is with NVIDIA CUPTI already available.

If one returns to the analogy of event logging being the software performance microscope, then this thesis provides three dimensional vision to this instrument. All layers of parallelism—inter-node communication, intra-node parallelism, and hardware accelerator utilization—are now observable concurrently.

Bibliography

[AMD11] AMD. *AMD APP Profiler User Guide*, Dec 2011. Online `http://developer.amd.com/tools/AMDAPPProfiler/html/index.html`, Download: Feb 26, 2012.

[AMD12] AMD. *AMD GPU Performance API*, Jan 2012. Online `http://developer.amd.com/tools/GPUPerfAPI/assets/GPUPerfAPI-UserGuide.pdf`, Download: Jun 25, 2012.

[BAB+08] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson. 0.374 Pflop/s trillion-particle kinetic modeling of laser plasma interaction on Roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 63:1–63:11, Piscataway, NJ, USA, 2008. IEEE Press.

[BDH+08] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of Roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1:1–1:11, Piscataway, NJ, USA, 2008. IEEE Press.

[BHJR10] Holger Brunst, Daniel Hackenberg, Guido Juckeland, and Heide Rohling. Comprehensive Performance Tracking with Vampir 7. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 17–29. Springer Berlin Heidelberg, 2010.

[BL95] Charles K. Birdsall and A. Bruce Langdon. *Plasma Physics via Computer Simulation (Plasma Physics Series)*. Inst. of Physics Publ., Bristol and Philadelphia, repr. edition, 1995.

[Bow01] K. J. Bowers. Accelerating a paricle -in-cell simulation using a hybrid counting sort. *J. Comput. Phys.*, 173(2):393–411, November 2001.

[Bri11] Encyclopædia Britannica. coprocessor. `http://www.britannica.com/EBchecked/topic/1366763/coprocessor`, 2011. Download: Jan 17, 2012.

[Bru07] Holger Brunst. *Integrative Concepts for Scalable Distributed Performance Analysis and Visualization of Parallel Programs*. PhD thesis, Technische Universität Dresden, 2007.

[Bru10] Holger Brunst. Vampir. Center for Information Service and High Performance Computing (ZIH), personal communications, 2010.

[Bus12] Michael Bussmann. PIConGPU. Helmholtz Zentrum Dresden-Rossendorf (HZDR), personal communications, 2008-2012.

[BWH+10] Heiko Burau, René Widera, Wolfgang Hönig, Guido Juckeland, Alexander Debus, Thomas Kluge, Ulrich Schramm, Tom Cowan, Roland Sauerbrey, and Michael Bussmann. PICon-

GPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. *Plasma Science, IEEE Transactions on*, 38(10):2831–2839, 2010.

[Cen07] Barcelona Supercomputing Center. BSC Instrumentation packages for Cell/B.E. Online `http://www.bsc.es/computer-sciences/programming-models/linux-cell/bsc-tools`, Download Mar 4, 2012, Nov 2007.

[Cle08] ClearSpeed. *Introductory Programming Manual – The ClearSpeed Software Development Kit*. ClearSpeed Technology Ltd, United Kingdom, Jan 2008. Revision 2E.

[Cle10] ClearSpeed. *Visual Profiler User Guide*. ClearSpeed Technology Ltd, United Kingdom, Sep 2010. Version 3.1, Document No. 06-RM-1136 Revision: 5.A.

[Cle11] ClearSpeed. *CSX700 Floating Point Processor Datasheet*. ClearSpeed Technology Ltd, United Kingdom, Jan 2011. Revision 1E.

[Con09a] Convey. Convey Compiler Datasheet. Online, `http://www.conveycomputer.com/Resources/Compiler%20Data%20Sheet.pdf`, Download: Feb 1, 2012, 2009.

[Con09b] Convey. Convey Personality Development Kit Datasheet. Online, `http://www.conveycomputer.com/Resources/PersonalityDevelopmentKit.pdf`, Download: Feb 1, 2012, 2009.

[Con10] Convey. Convey HC1ex Datasheet. Online, `http://www.conveycomputer.com/Resources/Convey_HC1_Family.pdf`, Download: Feb 1, 2012, 2010.

[Dem11] Eric Demers. Evolution of AMD Graphics. Presentation at AMD Fusion Developer Summit 2011, Online `http://developer.amd.com/afds/assets/keynotes/6-Demers-FINAL.pdf`, Download: Feb 2, 2012, Jun 2011.

[Die09] Robert Dietrich. SGI RASC: Evaluierung einer Programmierplattform zum Einsatz von FPGAs als Hardware-Beschleuniger im Hochleistungsrechnen. Master's thesis, Technische Universität Dresden, 2009.

[DIJ10] Robert Dietrich, Thomas Ilsche, and Guido Juckeland. Non-intrusive Performance Analysis of Parallel Hardware Accelerated Applications on Hybrid Architectures. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 135–143, Washington, DC, USA, 2010. IEEE Computer Society.

[EOO+06] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband EngineTM architecture. *IBM Syst. J.*, 45:59–84, January 2006.

[Fre11] Free Software Foundation. GNU Compiler Manpage, Oct 2011. GCC version 4.6.2.

[FWS10] Karl Fürlinger, Nicholas J. Wright, and David Skinner. Performance Analysis and Workload Characterization with IPM. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 31–38. Springer Berlin Heidelberg, 2010. 10.1007/978-3-642-11261-4_3.

[FWS11] K. Furlinger, N.J. Wright, and D. Skinner. Comprehensive Performance Monitoring for GPU Cluster Systems. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1377 –1386, may 2011.

[GWW+10] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22:702–719, April 2010.

[Hab08] Gad Haber. *Cell/B.E. SDK 3.0 tools, Part 1: Using performance tools*. IBM, Apr 2008.

[Hac07] Daniel Hackenberg. Einsatz und Leistungsanalyse der Cell Broadband Engine. Großer Beleg (Junior Thesis), Technische Universität Dresden, 2007.

[Hac08] Daniel Hackenberg. Über die Cell/B.E.-Architektur: Optionen zur Generierung von Programm-Traces. Master's thesis, Technische Universität Dresden, 2008.

[HB00] Jeffrey K. Hollingsworth and Bryan R. Buck. An API for Runtime Code Patching. *Journal of High Performance Computing Applications 14 (4)*, 2000.

[HD08] Scott Hauck and Andre DeHon, editors. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgen Kaufmann, 2008.

[HJB12] Daniel Hackenberg, Guido Juckeland, and Holger Brunst. Performance analysis of multi-level parallelism: inter-node, intra-node and hardware accelerators. *Concurrency and Computation: Practice and Experience*, 24(1):62–72, January 2012.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantative Approach*. Morgen Kaufmann, Fourth edition, 2007.

[HP10] Imran S. Haque and Vijay S. Pande. Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 691–696, Washington, DC, USA, 2010. IEEE Computer Society.

[HSW+10] Wolfgang Hönig, Felix Schmitt, René Widera, Heiko Burau, Guido Juckeland, Matthias S. Müller, and Michael Bussmann. A Generic Approach for Developing Highly Scalable Particle-Mesh Codes for GPUs. In *Proceedings of the 2010 Symposium on Application Accelerators in High Performance Computing*, SAAHPC '10, 2010.

[Hö12] Wolfgang Hönig. Towards Source-Level CUDA Kernel Profiling. Master's thesis, Technische Universität Dresden, Feb 2012.

[ia09] intel assembler.it. The 8087 Instruction Set. Online: `http://www.intel-assembler.it/portale/5/the-8087-instruction-set/a-one-line-description-of-x87-instructions.asp`, Download: Jan 19, 2012, Jun 2009.

[IHB11] Thomas Ilsche and Rebecca Hartman-Baker. *Vampir Introduction - Trace-based Performance Analysis*. Oak Ridge National Laboratory, Oct 2011.

[Ils09] Thomas Ilsche. Automatische Generierung von Programmspuren bei Bibliotheksaufrufen. Großer Beleg (Junior Thesis), Technische Universität Dresden, 2009.

[Int11] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Dec 2011. Document-No.: 325462-041US.

[ISC+12] Thomas Ilsche, Joseph Schuchart, Jason Cope, Dries Kimpe, Terry Jones, Andreas Knüpfer, Kamil Iskra, Robert Ross, Wolfgang E. Nagel, and Stephen Poole. Enabling Event Tracing at Leadership-Class Scale through I/O Forwarding Middleware. In *The 21 International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2012)*, Delft, The Netherlands, June 2012.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley- Interscience, New York, NY, Apr 1991.

[KBB+06] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In Vassil N. Alxandrov, Geert D. Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *6th International Conference on Computational Science (ICCS)*, volume 2, pages 526–533, Reading, UK, 2006. Springer.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49:589–604, July 2005.

[KDY11] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. GPU Application Development, Debugging, and Performance Tuning with GPU Ocelot. In Wen-Mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, pages 409–427. Morgan Kaufmann, 2011.

[KH10] David R. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, Morgen Kaufman, 2010.

[Khr11] Khronos Group. OpenCL Standard. Online `http://www.khronos.org/opencl`, Download: Feb 2, 2012, Nov 2011. Version 1.2.

[Klu11] Michael Kluge. *Hochleistungsrechnen mit Hardwarebeschleunigern*. Scientific Talk as part of the Ph.D. defense, Technische Universität Dresden, 2011.

[Knü08] Andreas Knüpfer. *Advanced Memory Data Structures for Scalable Event Trace Analysis*. PhD thesis, Technische Universität Dresden, 2008.

[Knü12] Andreas Knüpfer. C3G Data Structure and Algorithms. Center for Information Service and High Performance Computing (ZIH), personal communications, 2012.

[KRaM+11] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorf, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P – A joint performance measurement runtime infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of the Intl. Parallel Tools Workshop*, Sept. 2011. (accepted).

[Lab10] J. Labarta. New Analysis Techniques in the CEPBA-Tools Environment. In Müller, M. S., Resch, M. M., Schulz, A., & Nagel, W. E. , editor, *Tools for High Performance Computing 2009*, page 125. Springer, 2010.

[LANL] LANL Los Alamos National Laboratory. Roadrunner Website. Online: `http://www.lanl.gov/roadrunner`, Download: Jan 20, 2012.

[Lie12] Matthias Lieber. *Dynamische Lastbalancierung und Modellkopplung zur hochskalierbaren Simulation von Wolkenprozessen*. PhD thesis, Technische Universität Dresden, 2012.

[MBS+11] A.D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 176 –185, sept. 2011.

[MBSM10] Allen D. Malony, Scott Biersdorff, Wyatt Spear, and Shangkar Mayanglambam. An experimental approach to performance measurement of heterogeneous parallel applications using CUDA. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 127–136, New York, NY, USA, 2010. ACM.

[Mes95] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`, 1995. Version 1.1.

[MH11] Michael Mantor and Mike Houston. AMD Graphic Core Next. Presentation at AMD Fusion Developer Summit 2011, Online `http://developer.amd.com/afds/assets/presentations/2620_final.pdf`, Download: Feb 2, 2012, Jun 2011.

[Mic07] Holger Mickler. Kombinierte Messung und Analyse von Programmspuren und systemweiten I/O-Ereignissen. Master's thesis, Technische Universität Dresden, 2007.

[MK12] Kathryn Mohror and Karen L. Karavanic. Trace profiling: Scalable event tracing on high-end parallel systems. *Parallel Computing*, 38(4–5):194 – 225, 2012.

[MKJ+08] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In Christian Bischof, Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications*, volume 15 of *Advances in Parallel Computing*, pages 637–644. IOS Press, 2008.

[MMS10] Shangkar Mayanglambam, Allen D. Malony, and Matthew J. Sottile. Performance Measurement of Applications with GPU Acceleration using CUDA. In Barbara Chapman, Frédéric Desprez, Gerhard R. Joubert, Alain Lichnewsky, Frans Peters, and Thierry Priol, editors, *Parallel Computing: From Multicores and GPU's to Petascale*, volume 19 of *Advances in Parallel Computing*, pages 341–348. IOS Press, 2010.

[MMSH10] Alan Morris, Allen D. Malony, Sameer Shende, and Kevin Huck. Design and Implementation of a Hybrid Parallel Performance Measurement System. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, ICPP '10, pages 492–501, Washington, DC, USA, 2010. IEEE Computer Society.

[NAW+96] Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 1(12):69–80, 1996.

[Net] Data Direct Networks. S2A Technology. Online: http://www.ddn.com/products/s2a-technology, Download: Jan 23, 2012.

[NVI00] NVIDIA. NVIDIA To Acquire 3dfx Core Graphics Assets. Press Release, http://www.nvidia.com/object/IO_20010612_6602.html, Dec 2000.

[NVI08] NVIDIA. NVIDIA to Acquire AGEIA Technologies. Press Release, http://www.nvidia.com/object/io_1202161567170.html, Feb 2008.

[NVI11a] NVIDIA. *Compute Visual Profiler: User Guide*, May 2011. Version 4.0, Document ID: DU-05162-001_v04.

[NVI11b] NVIDIA. *CUDA C Programming Guide*, Nov 2011. Version 4.1.

[NVI11c] NVIDIA. NVIDIA, Cray, PGI, CAPS Unveil 'OpenACC' Programming Standard for Parallel Computing. Press release. Online http://www.openacc-standard.org/announcements-1/nvidiacraypgicapsunveil%E2%80%98openacc%E2%80%99programmingstandardforparallelcomputing, Download: Feb 2, 2012, Nov 2011.

[NVI11d] NVIDIA. Parallel Nsight User Guide. Online http://http.developer.nvidia.com/ParallelNsight/2.1/Documentation/UserGuide/HTML/Parallel_Nsight_User_Guide.htm, Download: Feb 26, 2012, 2011. Rev. 2.1.120113.

[NVI12a] NVIDIA. *CUDA C Best Practices Guide*, Jan 2012. Document ID DG-05603-001_v4.1.

[NVI12b] NVIDIA. *CUDA Tools SDK – CUPTI User's Guide*, Feb 2012. Document DA-05679-001_v03.

[Ope11a] OpenACC. The OpenACC Application Programming Interface. Online http://www.openacc-standard.org/Downloads/OpenACC.1.0.pdf, Download: Feb 2, 2011, Nov 2011. Version 1.0.

[Ope11b] OpenMP Architecture Review Board. OpenMP Application Program Interface. Online http://www.openmp.org/mp-documents/OpenMP3.1.pdf, Download: Feb 2, 2012, Jul 2011.

[Por11] The Portland Group. PGI Accelerator Compilers Website. Online http://www.pgroup.com/resources/accel.htm, Download: Feb 2, 2012, 2011.

[PSAW09] Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Henry Wong. Micro-benchmarking the GT200 GPU. Technical report, Computer Group, ECE, University of Toronto, 2009.

[Ros11] Ofer Rosenberg. OpenCL Overview. Online http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf, Download: Feb 2, 2012, Nov 2011.

[Saa03] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[SH11] G. Stoker and J.K. Hollingsworth. Towards a Methodology for Deliberate Sample-Based Statistical Performance Analysis. In *Parallel and Distributed Processing Workshops and*

Phd Forum (IPDPSW), 2011 IEEE International Symposium on, pages 1258 –1265, May 2011.

[Sha49] Claude Elwood Shannon. Communication in the Presence of Noise. *Proceedings - Institute of Radio Engineers*, 37(1):10–21, Jan 1949.

[Ski05] David Skinner. Performance Monitoring of Parallel Scientific Applications. Technical report, Lawrence Berkeley National Laboratory, May 2005. LBNL-PUB-5503.

[SM06] Sameer S. Shende and Allen D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, Summer 2006.

[SW81] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[TJYD10] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting Performance Data with PAPI-C. In Müller, M. S., Resch, M. M., Schulz, A., & Nagel, W. E. , editor, *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[TSV07] George Theodoridis, Dimitrios Soudris, and Stamatis Vassiliadis. A Survey of Coarse-Grain Reconfigurable Architectures and Cad Tools. In Stamatis Vassiliadis and Dimitrios Soudris, editors, *Fine- and Coarse-Grain Reconfigurable Computing*, pages 89–149. Springer, 2007.

[VG02] Frank Vahid and Tony Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, 2002.

[VMMR09] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Scalable I/O tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 26–31, New York, NY, USA, 2009. ACM.

[WBB12] Matthias Weber, Ronny Brendel, and Holger Brunst. Trace file comparison with a hierarchical sequence alignment algorithm. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 247 –254, july 2012.

[Wid12] René Widera. PIConGPU. Center for Information Services and HPC (ZIH) and Helmholtz Zentrum Dresden-Rossendorf (HZDR), personal communications, 2008-2012.

[WJMN09] Michael Wagner, Guido Juckeland, Matthias S. Müller, and Wolfgang E. Nagel. Comparing NVIDIA and AMD GPU Computing: Performance, Flexibility, and Usability. Presented at MRSC 2009, April 2009.

[ZLGS99] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward Scalable Performance Visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.

List of Figures

List of Tables