2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing

High Resolution Program Flow Visualization of Hardware Accelerated Hybrid Multi-Core Applications

Daniel Hackenberg, Guido Juckeland, and Holger Brunst
Center for Information Services and High Performance Computing (ZIH)
Technische Universität Dresden – 01062 Dresden, Germany
Email: {daniel.hackenberg, guido.juckeland, holger.brunst}@tu-dresden.de

***Abstract*—The advent of multi-core processors has made parallel computing techniques mandatory on main stream systems. With the recent rise of hardware accelerators, hybrid parallelism adds yet another dimension of complexity to the process of software development. This article presents a tool for graphical program flow analysis of hardware accelerated parallel programs. It monitors the hybrid program execution to record and visualize many performance relevant events along the way. Representative real-world applications written for both IBM's Cell processor and NVIDIA's CUDA API are studied exemplarily. To the best of our knowledge, this approach is the first that visualizes the parallelism in hybrid multi-core systems at the presented level of detail.**

I. Introduction and Background

Multi-core technology ultimately found its way into mainstream processor families. In order to fully utilize the potential of modern computer systems, parallel programming techniques have become mandatory for developers. The growing popularity of hardware accelerated computing further increases the complexity of the software development process. Hardware accelerators often offer one order of magnitude higher computing power, but this comes at the cost of moving from homogeneous to heterogeneous (or hybrid) parallelism. Especially the correct and efficient usage of the increasingly complex memory hierarchy is mandatory to obtain best application performance. Unfortunately, its inner-working is normally well hidden from the application developer. With traditional profilers, the impact of code tuning activities can often only be observed from a rather external point of view. The exact identification and location of the culprit is often impossible and the real reasons behind a performance deficiency remain obscure.

We decided to explain and verify our concepts for program execution analysis in the scope of IBM's Cell and NVIDIA's CUDA APIs due to their demanding programming models. Both approaches entered the HPC arena as product developments which were backed up by the computer gaming mass market. The limitation on these two products is a mere practical choice, the general observations and principles also hold true for other accelerators.

While there exist several major architectural differences, Table I reveals several similarities between the two accelerators and traditional multi-core CPUs [1], [2]. We introduce common terms for common architectural features to simplify further discussions in this paper. These similarities are important for our program monitoring approach that takes the complexity of a program execution in both environments into account (see Fig. 1).

There are also fundamental differences between the two architectures, namely how the accelerators access the memory of the host processor. While Cell allows direct remote memory access (also to other SPE's local memory), the CUDA API requires explicit data transfers between host and device memory. Therefore, Cell DMA instructions can be *accelerator-initiated* while CUDA `memcpy` instructions are *host-initiated*. Both are intended to transfer large amounts of data. Cell also offers a mechanism for small *mailbox messages* that are limited to 32 bits in size. A multi-core CPU provides a shared memory address space to all participating threads.

The information sources mentioned above are most valuable when designing or porting a (hybrid) multi-core application. They form the basis for the evaluation of three general performance tuning topics:

1) Memory utilization (Data transfers, global/host memory access)
2) Core/Unit utilization
3) Task/Thread synchronization

However, acquisition, processing, presentation, and interpretation of this information is not an easy task in many respects as explained in the following paragraphs. Existing tools need to be extended to deal with the new hardware and data complexity if a realistic assessment is intended.

Our combined monitoring and visualization approach makes the next step in tool evolution towards a highly improved level of detail, precision, and completeness. While our approach addresses a general problem we also demonstrate the practical aspects of it in a hybrid real world environment.

This paper is organized as follows: Section II outlines our software tracing design for hardware accelerators and the reference implementations for Cell and CUDA. A monitoring overhead discussion as well as several case studies for both the Cell and the CUDA platform are presented in Section III. We use real-life applications to demonstrate how

978-0-7695-4039-9/10 $26.00 © 2010 IEEE
DOI 10.1109/CCGRID.2010.27

Table I
COMPARISON OF ARCHITECTURAL FEATURES FOR THE IBM CELL, THE NVIDIA G200B GPU AND A GENERAL PURPOSE MULTI-CORE CPU

Feature	IBM Cell	NVIDIA GPU	Multi-core CPU
Host processor	PPE	CPU	one core
Accelerator	SPE	GPU	all cores
Local memory	256 KB local store	16 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 sychronization	shared address space
Accelerated program	SPE program	CUDA kernel	multi-threaded program

performance problems can be observed and analyzed with our new monitoring technology. Section IV concludes this paper and sketches future work.

II. MICROSCOPIC PROGRAM OBSERVATION

The advantages and disadvantages of sampling versus logging with respect to performance analysis of parallel applications have been well examined [3]. A general conclusion can be that tabular execution profiles are good at identifying dominant program sections at the cost of a constant overhead for acquiring the performance data. Especially for parallel applications more insight into execution of the identified code region is needed. The logging approach, also known as "event tracing", in combination with a sophisticated event browser enables a microscopic view on the temporal execution of parallel programs including the exact timing information of specific events.

A. Accelerator-aware Program Logging

Monitoring the host processor of an application that uses hybrid parallelism is straight forward as it can be handled with standard technology. One example is the Open Source software monitor VampirTrace [4] that we use to conduct the case studies presented in this article. With several new extensions, VampirTrace also monitors specific events of the two accelerator APIs that we focus on in this paper, namely IBM's Cell and NVIDIA's CUDA.

First of all, the thread creation on the respective cores needs to be detected and propagated to the host processor which acts as a control unit and supervisor. This includes the synchronization of the individual hardware timers across multiple processors and their cores. During program execution, data transfers, remote access, and user functions (kernels) are logged by customized wrapper libraries. Accelerated program sequences are logged by customized accelerator monitors. The fact that host processor and accelerator use disjoint memory regions is an additional challenge. Thus, log entries that are generated on the accelerator have to be transferred to the host memory and merged into the context of the execution log of the whole application.

The limited amount of local memory on Cell's SPEs requires sophisticated techniques such as double buffering and post-mortem log generation to keep program perturbation low [5]. GPUs, on the other hand, do not offer logging interfaces for local memory accesses. While they offer large amounts of device memory to capture time stamps for occurring events, the available performance critical events are currently limited to start- and end times of CUDA kernels and non-coalesced device memory accesses. This is due to the fact that more detailed performance hooks into CUDA are not yet publicly available.

B. Visualization

The program flow and performance graphics in this article were generated by the Vampir visualization tool [6]. For a better understanding, some aspects of the accelerator architectures have to be explained with respect to the applied visualization. Our general proposal for accelerated hybrid program flow visualization is outlined in Fig. 2. The host process and its accelerated parts are individually assigned to horizontal bars that change in color to reflect different program regions (e.g. function calls). An important point is

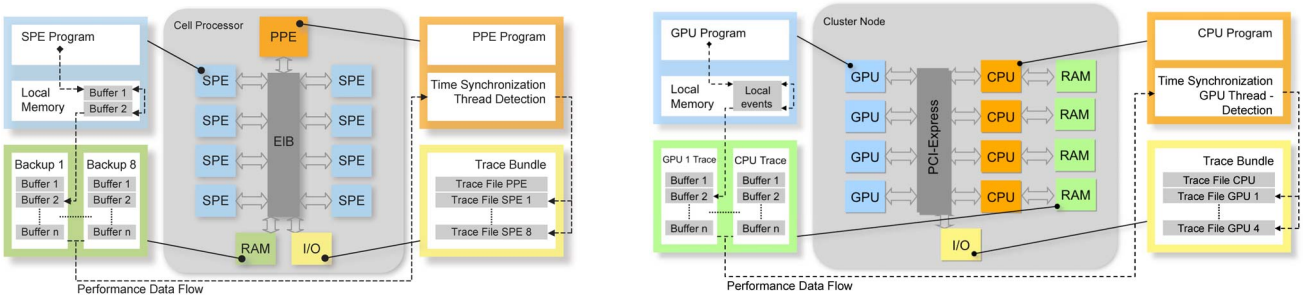


Fig. 1. Cell processor (left) and a GPU compute node (right) block diagram surrounded by the software monitor components for Host, Accelerator, Memory, and I/O. The log data flow is indicated by a dashed line.

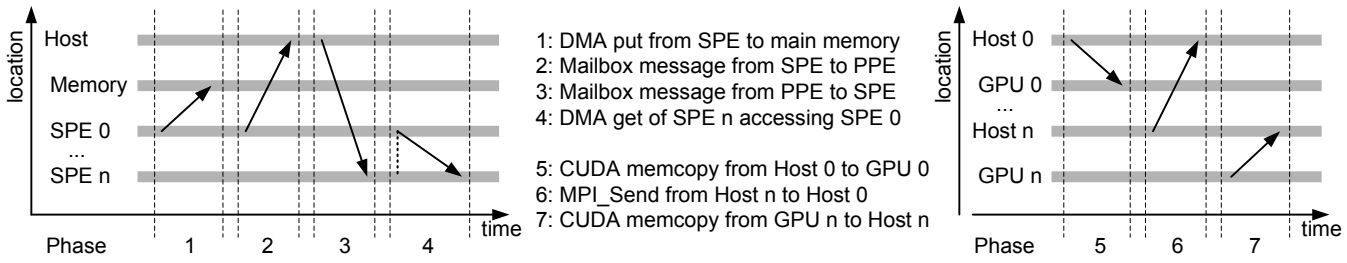


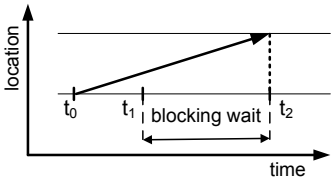
Fig. 2. Visualization of different Cell (left) and CUDA (right) communication primitives

to correctly display the target of a data transfer. Cell and GPU differ in this respect due to their different data transfer techniques as discussed in Section I.

A DMA access that is triggered on an SPE may access the main memory as well as another SPE's local memory. The latter case can be depicted as a connecting line between two SPE bars as illustrated in Phase 4 of Fig. 2. For the former case we introduce an additional horizontal bar that represents the main memory. Consequently, a main memory access of an SPE is illustrated by a line connecting the SPE and the memory bar (Phase 1). Lines between a PPE process and an SPE thread refer to mailbox communication between these two partners (Phases 2 and 3 of Fig. 2). A very similar visualization is used to depict CUDA data transfers between host and device memory (Phases 5 and 7). An application may of course use multiple host processors with hardware accelerators and traditional MPI parallelism for the host processor communication as depicted in Fig. 2 (right).

In contrast to typical MPI "two-sided" communication, a DMA transfer on the Cell processor is one-sided, i. e., starting of transfers and waiting for their completion happens on the same core. We therefore introduce a dotted vertical line that helps to identify the active and the passive partner of the transfer as depicted in Phase 4 of Fig. 2.

A number of host processor and accelerator communication routines (e.g. CUDA kernel invocations or Cell DMA operations) have an asynchronous nature. Their trigger functions are non-blocking and immediately return after the communication has been initiated. Each asynchronous communication has to be concluded by a proper wait command prior to using its payload. This wait command stalls until the transfer is finished or returns immediately if it has already been finished before. Therefore, we measure the time before (t_1) and after (t_2) the call. Figure 3 illustrates




```
t_0 = time();
nonblocking_put(...);
do_compute();
t_1 = time();
blocking_wait();
t_2 = time();
```

Fig. 3. Logging asynchronous events, e.g. data transfers between host processor and accelerator core

this concept. If t_1 and t_2 differ by more than one clock cycle, we can conclude that $t_2 - t_1$ reflects the real waiting time. Consequently, a bandwidth calculation based on t_0 and t_2 is correct (unless the wait command refers to multiple data transfers). Furthermore, we can introduce a new time period between t_1 and t_2 that represents the actual wait phase. If the wait call returned immediately ($t_2 - t_1 \leq 1$), we can conclude that the data transfer most likely finished before t_1. Bandwidth calculations based on those events can be inaccurate as they only indicate a lower bound of the physical transfer rate.

The hybrid character of Cell and GPU performance data often requires selective visualization, e. g., host processor only, accelerators only, or a visualization that filters data transfers. Vampir's hierarchical display capabilities for hybrid message passing/thread parallelization and its customizable event filters offer this functionality to visualize traditional (MPI) parallel application logs. They are as effective when confronted with performance data originating from multiple host processors and their accelerators.

III. From Theory to Praxis

The presented multi-core software monitor infrastructure was tested, verified, and benchmarked on a representative set of real-world and test applications. First, we briefly introduce the applied performance data browser with the discussion of the program log of a Cholesky factorization that we ran on an IBM QS21 Cell blade. The algorithm solves dense systems of linear equations and is highly optimized, achieving more than 80% of Cell's theoretical peak performance [7]. We introduce several other applications that run on Cell- or GPU-accelerated systems to demonstrate the power of this form of program analysis to detect and optimize typical performance bottlenecks.

The most commonly used display is the timeline as depicted in Fig. 4 (left). The control unit (PPE, running Process 0), eight accelerator cores (SPEs), and the main memory are displayed according to the description in Section II-B. We can identify a typical startup phase with all accelerator cores waiting for messages from the control unit. It is followed by a first computational stage and the beginning of a second computational stage. We can also identify short synchronization phases in between these stages. Many

Fig. 4. Cholesky factorization on Cell: timeline (left), message statistics (center), message profile (right). Since DMA transfers overlay the underlying timeline activity, a lot of communication can lead to a solid black box hiding everything underneath. To avoid this, the superposed horizontal arrows illustrate that there is communication hidden. The size of the black circle in the middle of the circle indicates the number of hidden lines.

typical aspects of the algorithm can be discovered with little effort by navigating and zooming into different phases of the program.

The monitor collects detailed information of every individual communication operation. The center frame in Fig. 4 shows the average duration of transfers, broken down by individual communication pairs. It can be used to analyze communication patterns and identify irregularities. The bar chart at the right provides a statistical representation of message properties, for example the number of transfers broken down by message sizes.

A. Tackling the Memory Wall

One of the most challenging problems we face on todays processors is the so called *memory wall* [8]. It refers to the growing disparity between computational power and memory performance, the latter being characterized by the two parameters *bandwidth* and *latency*. Both bandwidth and latency of memory chips improve at a much lower speed than processor speeds. Conventional CPUs provide built-in prefetching and caching mechanisms to address this problem. Accelerators typically provide a more fundamental approach: programmers need to manage all transfers between global (host) memory and local (accelerator) memory in software. This allows to conveniently overlap device memory accesses (and communication in general) with computation, thus hiding the memory latency. For algorithms with sufficiently predictable memory access patterns this is a very effective way to significantly reduce the latency problems. The challenge for accelerator-aware performance tools is to gather, process and display data that can help programmers to exploit these new possibilities.

Our performance monitor automatically logs phases when an accelerator core stalls due to waiting for a software-controlled memory access. This can be phases when GPUs are idle because `memcpy` operations have not finished. Likewise, phases when SPEs stall while waiting for DMA or mailbox transfers will be recorded. In turn, the visualization allows to intuitively find attractive targets for optimization of memory accesses. Figure 5 shows a parallel sparse matrix-vector multiplication on Cell. Calculations (`calc`) and stalls due to DMA transfers (`dma_wait`) can be easily identified. The unoptimized case (Fig. 5 left) even shows phases where not a single accelerator core accesses main memory.

We apply two optimizations that overlap DMA transfers with computation (double buffering) and improve the memory bandwidth (128 Byte buffer alignment). The result in Fig. 5 (center) shows no `dma_wait` phases anymore. Such an observation shows that a program is computationally bound. In a next step, an computational optimization is likely to improve the overall performance. We choose loop unrolling as it typically has a significant effect on SPE programs. After this optimization the algorithm is memory bound again as indicated by the numerous `dma_wait` phases in Fig. 5 (right). The visualization helps to quickly understand that another computational optimization would have no effect. Further improvements would require to optimize or minimize the memory access.

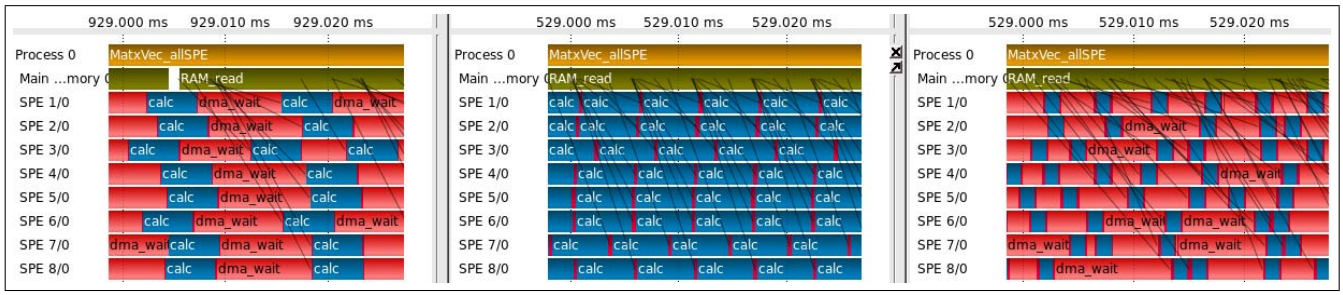


Fig. 5. Matrix vector multiplication on Cell: prior optimization (left), after memory optimization (center), and after computational optimization (right)

With software managed memory accesses on hardware accelerated systems, properly designed tools have the potential to resolve the obscurity of the memory wall that is usually generated by complex memory subsystems with hardware managed caches. In our case, the `dma_wait` phases allow programmers to actually "see" the memory wall and its effects within their applications.

It is important to note a unique feature of program logging compared to statistical approaches. Program logs allow to distinguish different stages as they occur in almost every real-life application. Take for example the Cholesky factorization depicted in Fig. 4 (left). The first stage of the algorithm (`ppu_convert_d2s`) is computationally bound while the second stage has a significant percentage of `dma_wait` phases. This insight can be easily gained with program logging and visualization. It is virtually impossible to achieve the same with a "sampling" approach.

B. Hybrid Analysis: Multi-Threading, Message Passing, and Accelerators

Hybrid programs that combine multiple parallelization paradigms like message passing and/or multi-threading with an accelerator library are still relatively rare. Their importance, however, has increased as hybrid HPC systems like Roadrunner [9] and large GPU cluster installations [10], [11] require this programming approach.

We use a scalable, parallel Particle-in-Cell (PIC) code [12] that was ported to CUDA as a showcase. Figure 6 shows the interaction of three GPUs and their corresponding host CPUs, which in turn also use MPI and pthreads to communicate and distribute work. Processes 1, 2 and 3 refer to the main program thread that transfers data between the GPU and the main memory. It is dominated by `CUDA_MEMCPY_SYNC` calls because the main program waits for the execution of the GPU kernels when calling a blocking operation such as `CUDA_MEMCPY_SYNC`. Threads

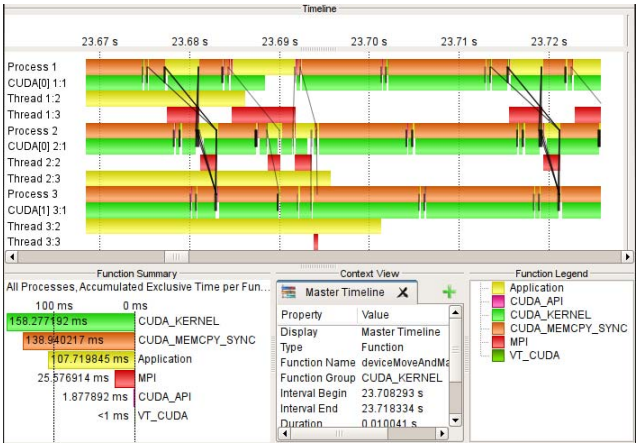


Fig. 6. Event log visualization of a 1024×3072 cells Particle-in-Cell (PIC) simulation run with three GPUs on two cluster nodes

1:2, 2:3 and 3:2 are responsible for visual data output of the GPU memory while threads 1:3, 2:2 and 3:3 are solely responsible for inter-node communication via MPI. The execution state of the CUDA kernels is visualized by the green bars named CUDA[0] 1:1, CUDA[0] 2:1 and CUDA[1] 3:1 where the number in brackets identifies the corresponding node.

The timeline display reveals that the execution of the CUDA kernels (and therefore the PIC computation on the GPUs) is barely interrupted by communication calls and the computational load on the GPUs is well balanced. This is the result of our performance optimization efforts that were facilitated by the program flow analysis presented in this paper.

C. Fine-grained Synchronization

Synchronization of parallel processes is among the effects that we intensely study using program logs and visualization. When we deal with multi-core processors, some parts of the traditional communication network move onto the chip. Message latencies are significantly lower and we can expect to see effects of much finer granularity.

Figure 7 (left) depicts a single, very short computational loop of another computational biology code (Randomized Accelerated Maximum Likelihood, RAxML) that has been ported to Cell [13]. The timeline display of just 4 μs shows the consecutive start of computations on each SPE. In fact, we can clearly see that the loop achieves only very limited parallelism due to the non-simultaneous start of the SPE computations. Figure 7 (right) shows the same timeline window after a small code modification that is intended to start SPE calculations more synchronously, which clearly is the case. However, we can also note that we introduced significant new memory contention because multiple SPEs access the main memory simultaneously. Therefore, the modification does not improve the overall RAxML performance. However, this example can serve as a showcase for our new analysis methods that can easily display runtime effects in the order of a few hundred nanoseconds.

D. Program Perturbation

Program logging generally introduces certain overhead sources. It is important to keep program perturbation at a minimum level to avoid destroying the runtime effects that we intend to study. For our approach this always requires a tradeoff between impact (runtime overhead) and outcome (number of events, visualization detail). Besides the number of events, the practical overhead is strongly determined by the concept and implementation of the event data acquisition and processing. Therefore, the performance impact on real-life applications is an important indicator for the quality of the monitor. Moreover, as the monitor implementations for different accelerator APIs may differ significantly, the program perturbation has to be determined individually.

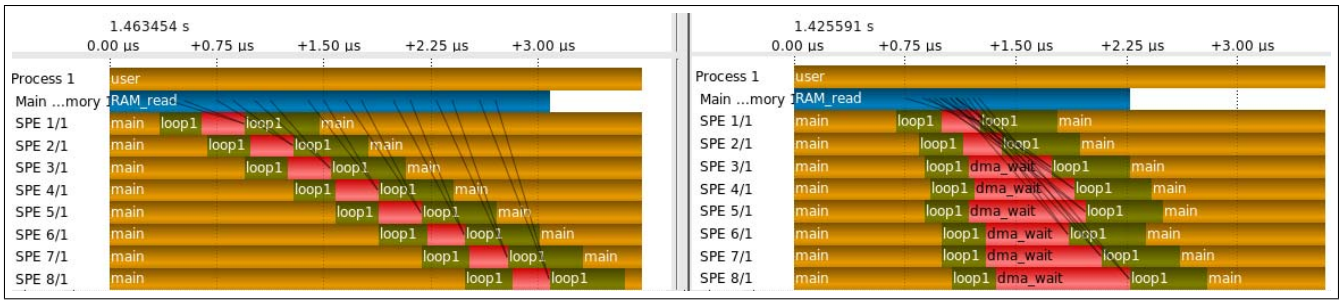


Fig. 7. Original (left) and modified (right) RAxML loop in a timeline window of 3.5 μs

On Cell, the transfer of SPE event buffers from local to main memory consumes memory bandwidth depending on the average event rate. Furthermore, the event buffers and the monitor itself need local memory that cannot be used by the application (currently less than 5%). If the trace events occur at a very high rate, event buffers may not be flushed quickly enough which would stall the SPE program until a buffer is available. SPE events also requires a function call into the trace library and the generation of the event log. Overall, the typical runtime overhead is below 3% [5].

For our CUDA monitor, all logging is currently done on the host processor. Thus, the impact on the kernel that runs on the GPU is minimal. Since the event recording is currently limited to kernel invocations, the log overhead is typically below 1% [14]. Future implementations that also generate events on the accelerator cores need to be designed in a way that keeps the overhead at a minimum level.

IV. Conclusion and Outlook

Modern (hybrid) multi- and many-core processors are a challenge to programmers who have to deal with an increasing amount of parallelism and complexity. This paper presents a software monitoring approach that provides developers with deep insights into their applications based on the recording and visualization of fine-grained performance logs. As proof of concept we have presented two monitoring implementations for IBM's Cell processor and CUDA applications for NVIDIA GPUs. Critical aspects like multithreading, asynchronous communication and memory operations, and the invocation of user functions can be studied individually over time and in the context in which they occur. Additionally, hybrid scenarios with MPI communication and heterogeneous processors are covered by a concept that keeps program perturbation at a low level.

New software approaches such as the OpenCL framework [15] abstract from the specific accelerator hardware to gain more hardware independence for program codes. OpenCL applications can for example run on Cell processors, ATI and NVIDIA GPUs, or multi-core CPUs and therefore offer a large portability advantage. We will focus our future work on supporting such frameworks in order to provide more developers with effective and efficient performance optimization tools.

References

[1] *Software Development Kit for Multicore Acceleration Version 3.0: Programmer's Guide*, IBM, 2007.

[2] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[3] E. Metz, R. Lencevicius, and T. F. Gonzalez, "Performance data collection using a hybrid approach," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 126–135, 2005.

[4] M. S. Müller et. al., "Developing scalable applications with Vampir, VampirServer and VampirTrace," in *Parallel Computing: Architectures, Algorithms and Applications*, 2007.

[5] D. Hackenberg, H. Brunst, and W. E. Nagel, "Event tracing and visualization for Cell Broadband Engine systems," in *Euro-Par*, 2008.

[6] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler, "Performance optimization for large scale computing: The scalable VAMPIR approach," in *International Conference on Computational Science*, 2001.

[7] J. Kurzak, A. Buttari, and J. Dongarra, "Solving systems of linear equations on the CELL processor using Cholesky factorization," *IEEE Transactions on Parallel and Distributed Systems*, 2007.

[8] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995.

[9] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: the architecture and performance of Roadrunner," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[10] Georgia Institute of Technology, "Georgia Tech wins NSF award for next-gen supercomputing," http://www.eurekalert.org/pub_releases/2009-10/giot-gtw102009.php, 2009.

[11] Y. Fei, and B. Ruixue, and W. Yushan, "Defense university builds china's fastest supercomputer," http://news.xinhuanet.com/english/2009-10/29/content_12356478.htm, 2009.

[12] C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation (Series in Plasma Physics)*, 1st ed. Taylor & Francis, October 2004.

[13] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos, "Dynamic multigrain parallelization on the Cell Broadband Engine," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2007.

[14] T. Ilsche, "Automatische Generierung von Programmspuren bei Bibliotheksaufrufen," TU Dresden - ZIH, Junior Thesis Paper, 2009, ZIH-R-0910.

[15] Khronos OpenCL Working Group, *The OpenCL Specification, Version: 1.0*, 2009.