Change Log

<table>
<thead>
<tr>
<th>Version</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0.1</td>
<td>Initial draft</td>
</tr>
<tr>
<td>v1.0</td>
<td>Final version including reviewer comments</td>
</tr>
</tbody>
</table>
## Contents

**Executive Summary** 5  

**1 Introduction** 6  

**2 OpenCL** 6  
- 2.1 Platform Model 6  
- 2.2 Execution Model 7  
- 2.3 Memory Model 8  
- 2.4 Profiling Interface 8  

**3 Score-P** 8  

**4 OpenCL Specific Performance Properties** 10  
- 4.1 Late allocation of HWAs 11  
- 4.2 Early releasing of HWAs 11  
- 4.3 Late data transfer 11  
- 4.4 Redundant data transfer 12  
- 4.5 Fragmented data transfer 12  
  - 4.5.1 Multiple memory transfers for one kernel 12  
  - 4.5.2 Scattered host copy 12  
- 4.6 Load imbalance 12  
  - 4.6.1 HWA is not utilized optimally 13  
  - 4.6.2 Host is not utilized optimally 13  

**5 OpenCL Support in Score-P** 13  
- 5.1 Library Wrapping in Score-P 14  
- 5.2 Usage of the OpenCL Profiling Interface 15  
- 5.3 Time Synchronization 16  

**6 Experiments** 16  
- 6.1 Data Visualization 16  
- 6.2 Profile Visualization with CUBE 16  
- 6.3 Trace Visualization with Vampir 17  
- 6.4 Simple Example on Mont-Blanc Prototype 17  
- 6.5 JuPedSim 17  

**7 Conclusion and Future Work** 20
Abbreviations

API  Application Programming Interface.
ARM  Advanced RISC Machines.
CPU  central processing unit.
CU   compute unit.
CUBE Profile browser and file format for profiles.
CUDA Compute Unified Device Architecture.
CUPTI CUDA Profiling Interface.
GPU  graphical processing unit.
HPC  High Performance Computing.
HWA  hardware accelerator.
JuPedSim Juelich Pedestrian Simulation.
MPI  Message Passing Interface.
OmpSs OpenMP SuperScalar programming model.
OpenCL Open Compute Language.
OTF2 Open Trace Format - version 2.
PE   processing element.
PMPI profiling interface for MPI.
Score-P Unified community measurement infrastructure.
Vampir Trace visualization tool.
Executive Summary

This document describes the work done to integrate basic support for the Open Compute Language (OpenCL) into the unified measurement infrastructure Score-P. After an introduction to OpenCL and Score-P the current status of the software prototype and preliminary results are presented in detail. The current prototype monitors important OpenCL API functions by intercepting them at link time and collecting the necessary data via library function wrapping. Data is captured on OpenCL functions regarding devices, kernels, memory objects and command queues. The prototype was tested with Intel, AMD and NVIDIA OpenCL implementations.
1 Introduction

The OpenCL programming model [1] enables developers to write or port their applications to accelerators, such as NVIDIA or AMD graphical processing units (GPUs) or the Intel Xeon Phi. In the course of the Mont-Blanc project applications are to be ported to an ARM-based system. OpenCL is one of the programming models to be investigated.

On a given hardware system, the main goal in High Performance Computing (HPC) is to achieve the highest possible performance. To achieve this goal a detailed understanding of all factors which have impact on the performance of an application is necessary. To this goal a number of software tools exist to measure, analyze and model the performance characteristics of applications. One of these tools is the Score-P performance measurement infrastructure, which is developed as a community project. Score-P performs instrumentation and measurement of applications on large-scale HPC systems. The measured data can be stored in form of profiles or traces and is then available for detailed visualization and analysis with the partners’ or third party developers’ tools.

To enable the Mont-Blanc project partners to evaluate their OpenCL porting efforts to the ARM-based prototype system, the integration of the respective support into Score-P was begun. The current prototype monitors important OpenCL API functions by intercepting them at link time and collecting the necessary data via library function wrapping. Data is captured on OpenCL functions regarding devices, kernels, memory objects and command queues. The prototype was tested with Intel, AMD and NVIDIA OpenCL implementations.

2 OpenCL

From the programmers point of view the hardware is managed by OpenCL in a hierarchy consisting of:

- **Platform**: abstraction of the underlying hardware.
- **Device**: in case of multiple accelerators belonging to the same platform, the device IDs allow to distinguish between them.
- **Context**: An abstraction consisting of:
  - **Devices**: Specific set of devices to be used
  - **Kernels**: Specific set of functions to be executed on these devices
  - **Program Objects**: Source code of the kernels
  - **Memory Objects**: Data on which the kernels operate

All of the above information is then used to create a command-queue object, which allows the scheduling of data transfers, kernel executions and synchronizations. The following sections describe the underlying abstractions of the platform model, the execution model and memory model. It concludes with the OpenCL profiling interface which is used in our measurement infrastructure to obtain performance data.

2.1 Platform Model

The smallest unit of a hardware accelerator (HWA) or a graphical processing unit (GPU) is a processing element (PE). Several of them are combined into a compute unit (CU). All PEs of
a CU are synchronized on the instruction level, thereby the necessary space on the chip can be tremendously reduced and a high level of parallelism can be achieved. Several of these CUs are combined into a device, which is connected to the host. Figure 1 shows the hierarchy as specified in OpenCL.

2.2 Execution Model

The main application is executed on the host. It manages all data transfers to the HWA and starts the execution of kernels. These are sets of instructions to perform calculations on the data previously copied to the device. After these calculations are finished the results are transferred back to the host. A kernel can be seen as a set of instructions that is executed multiple times on the HWA. To this end it is mapped onto the compute units, respectively processing elements:

- Processing element ↔ work item
- Compute unit ↔ work group

The index space of the data is mapped to the work items. The work items are combined into work groups. Work items are associated with unique global IDs. By mapping these IDs to indices of the data, each work item accesses the part of the data it is scheduled to work on. The work groups are also assigned an ID and each work item has a local ID inside its respective work group. Thereby a second way of mapping the data to the work items is made possible.
2.3 Memory Model

The memory on a HWA is classified into four different types:

- **Global memory:** All work items may access this memory with read and write instructions. Depending on the specific device, the operations may be cached.

- **Constant memory:** This is a region of the global memory that stores data which cannot be modified by an executing kernel. The host allocates the space and initializes it by copying the relevant data to it.

- **Local memory:** Each work group has exclusive access to its own local memory.

- **Private memory:** Each work item has exclusive access to its own private memory.

2.4 Profiling Interface

The OpenCL specification contains a profiling interface which can be used to obtain execution time of commands. To this end event objects are used which are passed as arguments to the following runtime functions:

- `clEnqueue[Read|Write|Map]Buffer`, `clEnqueue[Read|Write]BufferRect`,
- `clEnqueue[Read|Write|Map]Image`, `clEnqueueUnmapMemObject`,
- `clEnqueueCopyBuffer`, `clEnqueueCopyBufferRect`,
- `clEnqueueCopyImage`, `clEnqueueCopyImageToBuffer`,
- `clEnqueueCopyBufferToImage`, `clEnqueueNDRangeKernel`,
- `clEnqueueTask` and `clEnqueueNativeKernel`.

The function `clGetEventProfilingInfo` can then be used to query specific information about these events:

- **Time queued:** the current device time counter in nanoseconds when the command identified by event is enqueued in a command-queue by the host.

- **Time submitted:** the current device time counter in nanoseconds when the command identified by event that has been enqueued is submitted by the host to the device associated with the command-queue.

- **Time started:** the current device time counter in nanoseconds when the command identified by event starts execution on the device.

- **Time ended:** the current device time counter in nanoseconds when the command identified by event has finished execution on the device.

3 Score-P

The Score-P Instrumentation and Measurement Infrastructure [2] is a community-driven software framework for recording profiles and traces of parallel program execution. The application under investigation is automatically instrumented, by means of a number of different techniques, and linked to a set of libraries that implement the respective probe functions. Each invocation of a probe function is translated into measurement events such as enter/exit of code regions, or acquire/release of locks. Different metrics like number of visits, time spent in a region, bytes transferred over a network are associated with these events. Furthermore hardware counters
giving information about cache misses or floating point operations can be recorded. Figure 2 gives a general overview over the Score-P architecture.

Different methods for performing the instrumentation of an application are available. Many compilers allow for automatic instrumentation of user functions. At entries and exits of functions the compiler inserts probe functions and supplies source-code information. To instrument directive-based parallel programming paradigms, the source-to-source instrumentation tool OPARl2 is used. To examine communications that are accomplished with the Message Passing Interface (MPI), the profiling interface for MPI (P MPI) is used. It defines wrapper functions that are implemented as part of Score-P and allow to record MPI specific events and metrics. NVIDIA’s Compute Unified Device Architecture (CUDA) programming model is supported by means of the CUDA Profiling Interface (CUPTI).

Each of the aforementioned instrumentation techniques inserts different types of probe functions, which provide different types of information to the actual measurement system. To provide the measurement system with consistent data, Score-P contains a number of adapters. Each of these takes care of implementing the probe or wrapper functions for a specific kind of instrumentation.

There are two main modes of recording and storing data in Score-P: profiling and tracing. In a profile summarized data is recorded for each call-path executed by the program. Times and number of visits are aggregated; minimum, maximum and average values are stored. The values of performance counters are also recorded. In contrast in a trace every single instance of an event is recorded. This yields a very detailed view of the program run but comes at the cost of high memory demands during the measurement and in the storage of the trace file itself.
4 OpenCL Specific Performance Properties

There are many pitfalls for developers when adapting their applications to be run on a heterogeneous system. Due to the very different architecture of HWAs, compared to traditional central processing units (CPUs), programming techniques which have been developed and optimized in the last decades may no longer lead to an optimal solution. Performance properties that need to be considered can be put into one of three categories:

- Host-Accelerator Interaction
- Accelerator Memory Access Patterns
- Parallel Control Flow

The first category deals with the interaction between the host and the accelerator. The transfer of data to and from the HWA and the time it takes need to be considered. To make optimal use of the complete hardware, the workload should be distributed among host and HWA in a way that keeps them both occupied most of the time.

Second, the organization of the memory on accelerators is regarded. This differs from the memory hierarchies on conventional CPUs since there is a large number of processing elements which all perform the same instruction concurrently, i.e., all memory accesses take place at the same time. This leads to performance issues and needs special care of the programmer to ensure that the memory accesses are done in such a way that the different PEs do not hinder each other.

Finally, the tight synchronization of the PEs on the instruction level may lead to problems concerning the control flow. The amount of work scheduled on a compute unit may need too much memory for all the PEs of the corresponding CU to be utilized. Another factor that would lead to PEs not being used can be that the number of work items do not distribute evenly to the number of processing elements. Furthermore, HWAs are not able to deal well with code branching. Each time different threads branch to different code regions, dependent on the data they are working on, the code regions are executed serially. In the worst case this may lead to every PE performing work on its work item while all others wait. This leads to a loss of parallelism and thereby the main advantage of heterogeneous architectures is not exploited.

Performance analysis tools are needed more than ever to give programmers insight into the way they utilize their hardware. These tools must reliably detect a wide range of performance properties, determine their impact on the overall performance and give the user a good basis for optimizing their application.

In the case of OpenCL the data that can be collected with the profiling interface is limited to the first category: "Host-Accelerator Interaction". The other two categories are highly hardware dependent and we propose to employ hardware specific tools provided by the vendors to get detailed information about hardware usage.

The remainder of this section lists performance properties that may be collected or derived from information that is given by the profiling interface of the OpenCL runtime. In the following list each entry consists of a short description of the problem, a method to detect the property (including the required data that needs to be collected in a profile or trace file) and a way to measure the severity of the property. The severity is needed to distinguish the normal occurrence of the properties (it is to be expected that they can be found in every program) from a true performance-relevant problem.
4.1 Late allocation of HWAs

The first time a HWA is used, it needs to be allocated by the host. This allocation process takes some time and should be done asynchronously, i.e., as soon as possible in such a way that the host can perform its own setup and initialization at the same time as the allocation of the HWA is done.

**Detection:** A synchronous allocate event with the state START can always be treated as an instance of late allocation. For asynchronous allocate events, the time between the WAIT and STOP event is taken into account.

**Severity:**
- Synchronous: the time between the START and STOP event
- Asynchronous: the time between the WAIT and STOP event

4.2 Early releasing of HWAs

As described above the allocation of an HWA is time consuming. For this reason, releasing an HWA which might be needed again should be avoided, even though the second allocation can be much faster.

**Detection:** Release event which is followed by an allocate event. The two events do not necessarily have to be directly consecutive.

**Severity:**
- Synchronous: the sum of the times between the START and STOP event of the release and the unnecessary allocate event
- Asynchronous: the sum of the times between the WAIT and STOP event of the release and the unnecessary allocate event

4.3 Late data transfer

The transfer of data from the host to the accelerator is much slower than conventional memory access. Therefore, care should be taken to start the process of copying data as soon as possible, preferably asynchronously.

**Detection:** Synchronous data transfer can always be treated as an instance of late data transfer. For asynchronous transfer events, the time between the WAIT and STOP is taken into account.

**Severity:**
- Synchronous: the time between the START and STOP event
- Asynchronous: the time between the WAIT and STOP event

4.4 Redundant data transfer

Data which is needed by more than one kernel can be kept on the HWA. Two cases can be distinguished: constant data or modified data. In the case of constant data, the data is not modified. An identical copy resides on the host and is copied to the accelerator multiple times for each execution of the kernel. In the other case, the data is modified by one kernel and even though it will be used as input for the following kernel, it is copied back to the host only to be copied to the HWA again. Clearly this is very inefficient.
Detection: To detect this behavior the buffers would have to be monitored on the host side. For constant data, the multiple transfer of the same address space without the data having been accessed with a write operation, needs to be verified. In the case of the modified data, the same monitoring would have to be done for the buffer that is used to copy the data back from the HWA to the host. If this buffer is not modified on the host and used to copy data back to the HWA, redundant data transfer occurred.

Severity:

- Constant data: time for all unnecessary copying procedures from the host to the HWA
- Modified data: sum of the times for copying the data from HWA to host and back

4.5 Fragmented data transfer

There are two reasons for fragmented memory transfer: multiple memory transfers initiated by the programmer and data residing in separate areas of the memory of the host being copied in a single request. The problem is that each single transfer is associated with a certain overhead for the initialization of the communication. If all the data is packed into a single, continuous buffer, the overhead is reduced to a single instance.

4.5.1 Multiple memory transfers for one kernel

Most calculations depend on a number of variables. Transferring every variable in a separate operation causes a large overhead. If all the variables are copied into a single buffer on the host side and this buffer is transferred in one operation, the data only has to be separated again on the HWA side. This drastically reduces communication overhead.

Detection: Multiple instances of memory transfer events that are followed by the execution of a kernel.

Severity: Product of the estimated overhead of a single transfer operation and the number of excessive transfers.

4.5.2 Scattered host copy

Even when there is only a single transfer in the user code, the data may be stored in such way in the main memory of the host that it is not resident in a contiguous buffer but distributed over several locations. In this case, the memory transfer is split into multiple smaller transfers. It would be more efficient to first make a local copy on the host into one continuous buffer and then transfer the data in one block to the HWA.

This kind of storage can, for example, be found when the programmer first allocates an array of pointers and then allocates memory consecutively for each of these pointers. When allocating N-dimensional arrays in this way it is not guaranteed that the main data is actually in one consecutive block of memory.

Detection: There are multiple memory accesses before the execution of a kernel.

Severity: Product of the estimated overhead of a single data transfer and the number of data transfers.

4.6 Load imbalance

Load imbalance is a well known problem in traditional parallel architectures utilizing standard CPUs that are connected by a communication network. The introduction of GPUs as hardware accelerators introduces a new dimension to this problem. The main issue is that the performance
of the host and the HWA is different and the problem cannot just be divided into equal parts. Furthermore, the algorithms used on the host and HWA can also be very different resulting in varying data representations. This leads to more complicated algorithms and results in increased development costs. Unfortunately, not using the host while performing calculations on the HWA, or vice versa, is not an option as this would lead to a large amount of computing resources lying idle.

4.6.1 HWA is not utilized optimally

The workload is distributed in such a way that the HWA finishes before the host is done with its part of the calculation. This results in a waste of resources on the HWA.

**Detection:** The kernel finishes execution on the HWA before the host is ready to collect the results.

**Severity:** The time the HWA is idle multiplied by the performance of the HWA.

Let $W$ be the total workload, $p_H$ denote the performance of the host, $p_A$ the performance of the accelerator and $t_H$ be the time the host calculation took in the case of the load imbalance, then the severity would be calculated by:

$$ S = p_A(t_H - \frac{W}{p_H + p_A}) $$

(1)

This is of course highly theoretical as the workload normally cannot be determined exactly, especially if the number of iterations for the conversion is dependent on the input data. A more convenient estimate would be to simply take the difference of the time of execution on the host and the time of execution on the accelerator $t_A$:

$$ S = p_A(t_H - t_A) $$

(2)

4.6.2 Host is not utilized optimally

The workload is distributed in such a way that the host finishes before the HWA is done with its part of the calculation. This results in a waste of the resources of the host. Detection: The host finishes its part of the execution before the HWA is done with the kernel and is ready for the data to be copied back to the host.

**Severity:** The severity could be calculated exactly as:

$$ S = p_A(t_A - \frac{W}{p_H + p_A}) $$

(3)

Again, this is only theoretically possible. The proposed estimate is:

$$ S = p_H(t_H - t_A) $$

(4)

5 OpenCL Support in Score-P

To integrate OpenCL support into the measurement infrastructure Score-P a new adapter was created. The instrumentation process was modified so that at link-time this new adapter is now also linked to the instrumented application. A generic approach for creating wrapper functions was implemented and used specifically to perform library interposition of all OpenCL functions that are of interest for performance analysis. By wrapping these functions and using
the OpenCL profiling interface the allocation and deallocation of OpenCL devices and contexts is monitored. A similar approach was previously used in VampirTrace [3].

Detailed information about the time spent waiting in the command queue and on the accelerator, as well as time actually spent executing the data transfer or kernel executions is collected. The collected data is stored in the Score-P data formats CUBE and/or OTF2 (Open Trace Format - version 2), depending on whether profiling or tracing is performed. The following sections describe this approach in more detail.

5.1 Library Wrapping in Score-P

One approach to collect data about the usage of certain library functions in an application is to intercept these functions with a wrapper function. This wrapper function is then able to transform the information it has access to, i.e., function name and parameters, into a format that may be used by the measurement system. There the actual profiling or tracing is performed by keeping track of the time, number of visits and other relevant data, e.g., number of bytes transferred for MPI communication.

When linking an application there is a distinction between static and dynamic linking. In the case of static linking all functionality is directly included in the executable, there are no outside dependencies. When dynamic linking is used, the code to be executed in library function calls is located in a separate file that needs to be available during the runtime of the application.

```c
CL_INT SCOREP_LIBWRAP_FUNC_NAME( clGetPlatformInfo )
{
    CL_INT ret;

    SCOREP_OPENCL_FUNC_ENTER( scorep_opencl_region__clGetPlatformInfo );

    ret = SCOREP_LIBWRAP_FUNC_CALL( lw , clGetPlatformInfo ,
        ( cl_platform_id platform ,
        cl_platform_info paramName ,
        size_t paramValueSize ,
        void* paramValue ,
        size_t* paramValueSizeRet )
    );

    SCOREP_OPENCL_FUNC_EXIT( scorep_opencl_region__clGetPlatformInfo );

    return ret;
}
```

Listing 1: Example for the usage of the generic library wrapping mechanism in Score-P by usage of macros. Depending on the preselection of static or dynamic linking, different symbols are substituted for the actual function definition and call.

The generic library wrapping interface in Score-P provides the possibility to define and link wrapper functions to intercept the calls to the library functions. This is done via macros that
can be employed in the adapter. Depending on the linking approach that Score-P is configured with during its installation, the dynamic or static approach is automatically selected. Listing 1 shows how to use the macro `SCOREP_LIBWRAP_FUNC_NAME` to create a wrapper for the function `clGetPlatformInfo`. The actual measurement is performed during `SCOREP_OPENCL_FUNC_ENTER` and `SCOREP_OPENCL_FUNC_EXIT` while the actual library function of the OpenCL runtime is called with the macro `SCOREP_LIBWRAP_FUNC_CALL`.

5.2 Usage of the OpenCL Profiling Interface

As described in section 2.4 events are used by the OpenCL runtime to store performance data, like the time spent waiting for execution or the start and end timestamps of the operation. The respective function calls (e.g., to schedule a kernel) can be called with an event object or `NULL` as argument. In case the user requires an event object, e.g., to perform asynchronous execution and to be aware of the completion of a kernel execution, this object may also be used by the measurement system to retrieve the performance data. In the other case (when `NULL` is passed as argument) the OpenCL adapter in Score-P needs to create and manage an event object.

The usage of accelerators in OpenCL is managed with command-queues, which are defined by contexts (see 2). To store the collected performance data in a way that is meaningful to the user, data structures were implemented in Score-P that reflect these abstractions:

- Command queues
- Data transfers
- Kernel executions

These data structures are used to store all relevant information about the applications OpenCL usage. The measurement core of Score-P is not aware of these data structures, so they are then translated into the internal abstractions of Score-P. This is done via the `clGetEventProfilingInfo` function as described in section 2.4. The timing data is used to create events for profiling or tracing, by calling the generic functions in Score-P for entering and exiting of code-regions. This introduces some overhead which may potentially impact on the measured performance. Users of Score-P needs to be aware of this and fine-tune their measurements by filtering out regions that cause too much overhead and are not of direct interest.

As memory is a very restricted resource for applications, care needs to be taken to not interfere with the main applications memory requirements. The measurement system needs to allocate some memory, but there exists a strict limit of memory Score-P may allocate for recording performance data. Within this memory, the OpenCL adapter stores the OpenCL performance data, in form of event objects, in a buffer of fixed size. As described above, translating this event data to performance data in the format of the measurement core introduces overhead, which needs to be minimized.

The approach that was chosen, was to preferably use synchronization points between the host and the device, in which the host usually spends some time waiting for the executions on the device to complete. In this time the intermediate event buffers are read out by using the OpenCL profiling interface and the respective events are triggered in the measurement system to store profiling and/or tracing data. However, if the intermediate buffers of the adapter are fully used, a flushing operation is triggered before scheduling a new kernel or data transfer to avoid the loss of data. The user is informed of this via a warning during the runtime of the application.
5.3 Time Synchronization

The clocks on the host and the accelerator are not automatically synchronized. Furthermore there may be a drift between the different clocks. A linear model is used to calculate times in host-time ($t_h$) from the times in device-time ($t_d$). To this end timestamps are stored during the creation of a queue in host and device time ($t_{d\text{-}start}$ and $t_{h\text{-}start}$). When a stream of OpenCL events is translated into Score-P events another pair of host and device timestamps is taken ($t_{d\text{-}end}$ and $t_{h\text{-}end}$). To calculate host times $t_h$ from device times $t_d$ the following equation is used:

$$t_h = t_d \ast \frac{t_{h\text{-}end} - t_{h\text{-}start}}{t_{d\text{-}end} - t_{d\text{-}start}}$$

6 Experiments

To validate the protoypical integration of support for OpenCL into the measurement infrastructure Score-P experiments were carried out on two different systems. The first is the ARM-Exynos-based MontBlanc prototype cluster. The MontBlanc applications that run on this machine are ported with the OpenMP SuperScalar programming model (OmpSs)[4]. While this may use OpenCL underneath, combined support for both OpenCL and OmpSs is not yet integrated in Score-P. Therefore only very small OpenCL examples were used to show principal functionality on the ARM based machine.

A real application that was measured, running on an Intel Sandy-Bridge test-system with NVIDIA K40 GPUs, is the Juelich Pedestrian Simulation (JuPedSim) [5]. It is an agent based application that is used simulate evacuation scenarios of buildings. It was recently ported to OpenCL in the scope of a master’s thesis. A detailed measurement and performance evaluation of this approach was carried out, to show the applicability and value of the integration of OpenCL support into Score-P.

This section begins with a very brief introduction to the profile and trace visualization tools CUBE and Vampir that were used to visualize the collected performance data. These tools are then used to present the results of the experiments on the MontBlanc prototype and the performance analysis of JuPedSim.

6.1 Data Visualization

There are two main modes of collecting data with Score-P: profiling and tracing. While profiles contain aggregated data that are stored in the CUBE data format, traces in the OTF2 format contain information about every event on all processes, threads and devices. Different tools for the exploration of these data formats are available, the CUBE profile explorer and the Vampir trace browser.

6.2 Profile Visualization with CUBE

Figure 3 shows a visualization of a profile measurement of JuPedSim with CUBE [6]. The main view is divided into three panels. Metrics are presented in the leftmost panel. Selecting a metric (time in this case) reflects on the middle and right panel. The middle panel shows the call-tree of the program execution. The different nodes represent different regions in the code, in this case user functions that were instrumented with the compiler, OpenMP regions and OpenCL function calls that were profiled with the wrapping approach described in section 5.1. The leftmost panel shows the system tree, where the machine, the nodes, the processes, and the
threads on accelerator devices are shown in a hierarchical view. A selection in the middle or call-tree pane again determines the presented values in the system-tree pane.

6.3 Trace Visualization with Vampir

Figure 4 shows the main view of Vampir [7]. The view is highly configurable, in this case the window is divided into four views. The upper left view presents time-lines, one for each thread and the accelerator-device. Different regions of the code are color-coded, the legend in the lower right view gives an overview. The black lines in the time-line view show data transfers between the host and the device. The lower left view gives a summary of activities the processes perform for the selected time-frame. A complete accumulation over all processes is shown in the upper right window.

6.4 Simple Example on Mont-Blanc Prototype

As no "real" application was available on the Mont-Blanc prototype at the time of the first tests, a small example from the OpenCL SDK was used. The application performs the addition of two vectors on an accelerator and copies back the results. The Score-P prototype with OpenCL support was used to successfully instrument, build, and run this example. The results are visualized in Figures 5 and 6.

6.5 JuPedSim

JuPedSim is an open-source framework for performing pedestrian simulations, which is developed at the Juelich Supercomputing Centre. The primary goal of JuPedSim is to provide students and researchers with a toolbox that will let them focus on their main tasks, i.e. the development, calibration, and validation of new models or model features. JuPedSim is currently focusing on evacuation, but easily extendable to cover other areas e.g. passengers exchange, commuter traffic in railway stations etc.

It consists of four modules which are loosely coupled and can be used independently at the moment. These are:
The analysis performed in this work focuses on the core module, JPScore, where the actual simulation is performed. The main iteration loop consists of calculating repulsive forces between pedestrians, between pedestrians and walls, and updating velocity and positions. The calculation of forces between the pedestrians was ported with OpenCL to be scheduled on an accelerator device.

The JPScore module was instrumented and measured with Score-P. The tests were performed on a small test system in Juelich consisting of a host with two Intel\textsuperscript{®} Xeon\textsuperscript{®} CPU E5-2650 with together 16 cores (32 including hyper-threading) and 64GB RAM. Attached are two NVIDIA Tesla K40 with 12GB RAM. The experiments used 8 threads and one GPU.

As can be seen from the profile in figure 7, most of the time (3.67s) spent in the function clEnqueueWriteBuffer, which manages the data transfer of the input data that is necessary for the kernel execution to the device. The actual kernel execution time of 0.42s is comparatively small. In the system tree(right pane) it can be seen that the total execution time spent in the master thread for the data transfer and setup of kernel execution totals to 5.76s. This shows that there is some potential for optimization. When regarding the number of visits to the regions of the data transfers and the kernel executions (not shown in this screenshot) it becomes obvious that there are multiple data transfers for every kernel execution.

When inspecting the same region of the code in a trace with the Vampir trace viewer (figure 4) the data transfers and the kernel executions can be examined in some more detail.
Figure 5: CUBE Visualization of simple OpenCL test on the Mont-Blanc ARM-Exynos prototype

Figure 6: Vampir Visualization of simple OpenCL test on the Mont-Blanc ARM-Exynos prototype
Especially it becomes obvious that the input data for the kernel is transferred in multiple distinct data transfers. This corresponds to the performance issue described in section 4.5.1 "Multiple memory transfers for one kernel". Furthermore the usage of OpenCL is synchronous, it might be possible to use asynchronous execution to share work between the host and the device (see section 4.6).

It can be concluded that there is a direct potential for optimization. Only porting a single kernel with OpenCL is not sufficient to exploit the accelerator hardware. Close examination of the performance data with CUBE and Vampir shows optimization potentials that can be gained by distributing the work between host and device in a way that minimized data transfers and optimizes the total execution time.

7 Conclusion and Future Work

The measurement infrastructure Score-P was extended to support the OpenCL programming model. The OpenCL profiling interface was used to automatically and efficiently obtain performance data. The generic Score-P library wrapping approach was used to get access to data about used OpenCL functions. A proof of concept was shown on the target Mont-Blanc platform. On a small test system, an application that is used in production runs, but for which the port to OpenCL was only begun, was examined and valuable insights were gained, which will be used in the future. However the OpenCL support in Score-P remains prototypical and not all information that might be useful is yet recorded and made available to users. Furthermore the combined analysis of OmpSs application and OpenCL applications is not yet possible.

In the future the support for OpenCL will be more widely tested, therefore making it more stable and bringing it up to production quality code. Application developers in Mont-Blanc use OmpSs to automatically produce OpenCL code. To take this into account the two development branches of Score-P, for support of OpenCL and OmpSs, need to be unified to give users full information about the behavior of their application. At the moment it cannot be directly determined how useful more detailed information about OpenCL contexts and command queues would be. This needs to be investigated and if deemed important, integrated into Score-P.
References


