Mitglied der Helmholtz-Gemeinschaft

JÜLICH FORSCHUNGSZENTRUM

Debugging and performance analysis tools

November 22, 2017 | Ilya Zhukov

Outline

- Debugging
- Performance analysis: introduction
- Performance analysis tools
  - Score-P/CUBE
  - Vampir
  - AMD CodeXL
  - nvvp
  - Intel Vtune
  - Intel Advisor
- Tools overview
- Hands-on

Debugging OpenCL 1.1

- Top tip:
    - Write data to a global buffer from within the kernel

      result[get_global_id(0)] = … ;
    - Copy back to the host and print out from there or debug as a normal serial application
    - Works with any OpenCL device and platform

Debugging OpenCL

- **Check your error messages!**
    - If you enable Exceptions in C++ as we have here, make sure you print out the errors.
- Don't forget, use the err_code.c from the tutorial to print out errors as strings (instead of numbers), or check in the cl.h file in the include directory of your OpenCL provider for error messages
- Check your work-group sizes and indexing

Debugging OpenCL – using GDB

- Can also use GDB to debug your programs **on the CPU**
  - This will also leverage the memory system
    - Might catch illegal memory dereferences more accurately
  - But it does behave differently to accelerator devices so bugs may show up in different ways
- As with debugging, compile your C or C++ programs with the **–g** flag

Using GDB with AMD

- Ensure you select the CPU device from the AMD® platform
- Must use the –g flag and turn off all optimizations when building the kernels:

  program.build(" –g –O0")
- The symbolic name of a kernel function "**__kernel void foo(args)**" is "**__OpenCL_foo_kernel**"
  - To set a breakpoint on kernel entry enter at the GDB prompt:

    break __OpenCL_foo_kernel
- Note: the debug symbol for the kernel will not show up until the kernel has been built by your host code
- AMD® recommend setting the environment variable CPU_MAX_COMPUTE_UNITS=1 to ensure deterministic kernel behavior

Using GDB with Intel

- Ensure you select the CPU device from the Intel® platform
- Must use the –g flag and specify the kernel source file when building the kernels:

  program.build(" –g –s /full/path/to/kernel.cl")
- The symbolic name of a kernel function "**__kernel void foo(args)**" is "**foo**"
  - To set a breakpoint on kernel entry enter at the GDB prompt:

    break foo
- Note: the debug symbol for the kernel will not show up until the kernel has been built by your host code

Debugging OpenCL – using GDB

- use ***n*** to move to the next line of execution
- use ***s*** to step into the function
- if you reach a segmentation fault, ***backtrace*** lists the previous few execution frames
  - type ***frame 5*** to examine the 5th frame
- use ***print varname*** to output the current value of a variable

Performance factors of parallel applications

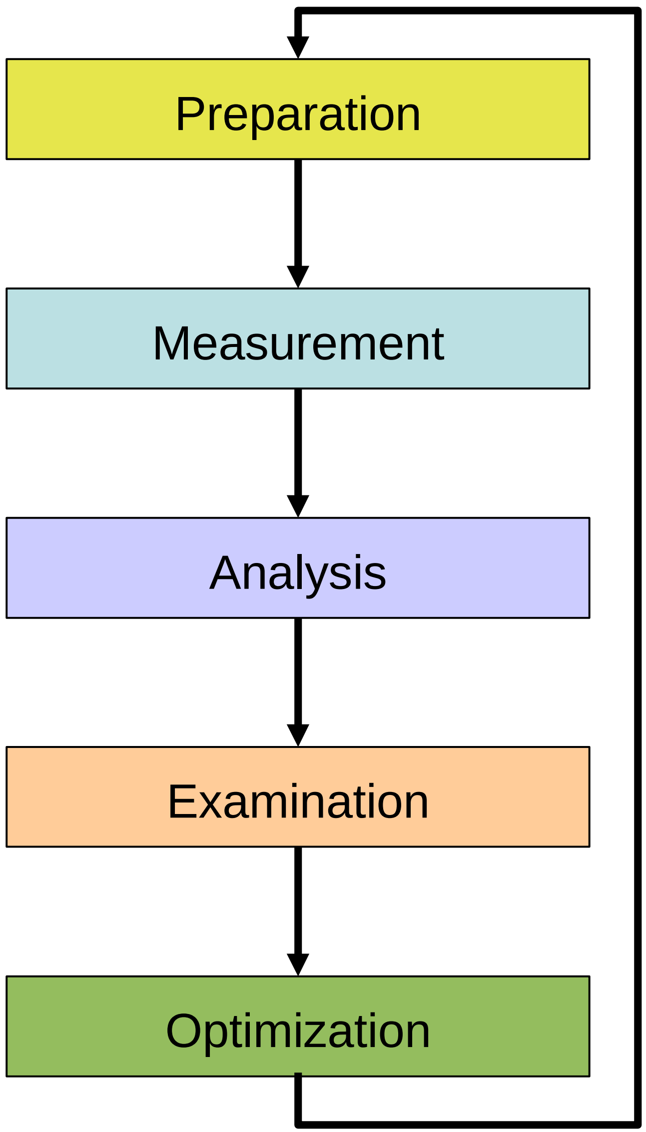
- "Sequential" factors
  - Computation
    - ✦Choose right algorithm, use optimizing compiler
  - Cache and memory
    - ✦Tough! Only limited tool support, hope compiler gets it right
  - Input / output
    - ✦Often not given enough attention

- "Parallel" factors
  - Partitioning / decomposition
  - Communication (i.e., message passing)
  - Multithreading
  - Synchronization / locking
    - ✦More or less understood, good tool support

Tuning basics

- Successful engineering is a combination of
  - The right algorithms and libraries
  - Compiler flags and directives
  - Thinking !!!
- Measurement is better than guessing
  - To determine performance bottlenecks
  - To compare alternatives
  - To validate tuning decisions and optimizations
    - ✦After each step!

Performance engineering workflow



- Prepare application (with symbols), insert extra code (probes/hooks)
- Collection of data relevant to execution performance analysis
- Calculation of metrics, identification of performance metrics
- Presentation of results in an intuitive/understandable form
- Modifications intended to eliminate/reduce performance problems

The 80/20 rule

- Programs typically spend 80% of their time in 20% of the code
- Programmers typically spend 20% of their effort to get 80% of the total speedup possible for the application
    - *Know when to stop!*

- Don't optimize what does not matter
    - *Make the common case fast!*

> “If you optimize everything, you will always be unhappy.”
>
> Donald E. Knuth

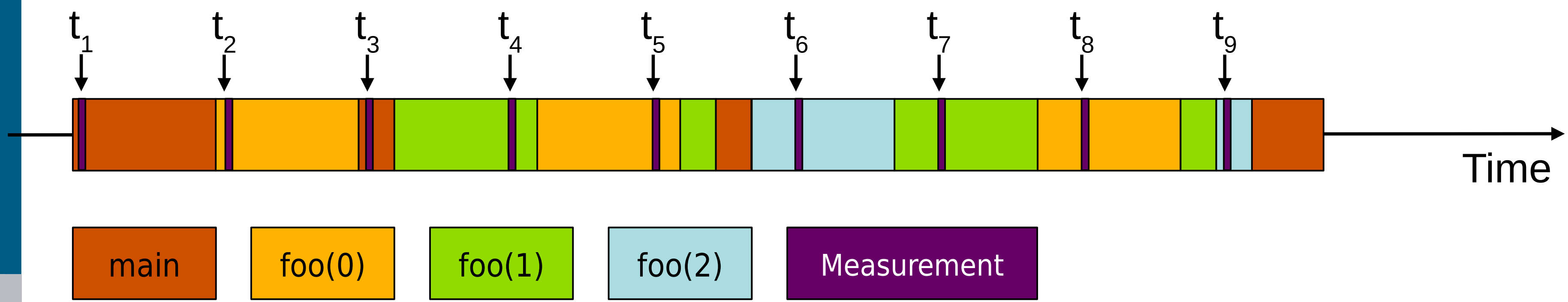
Classification of measurement techniques

- How are performance measurements triggered?
  - Sampling
  - Code instrumentation

- How is performance data recorded?
  - Profiling / Runtime summarization
  - Tracing

- How is performance data analyzed?
  - Online
  - Post mortem

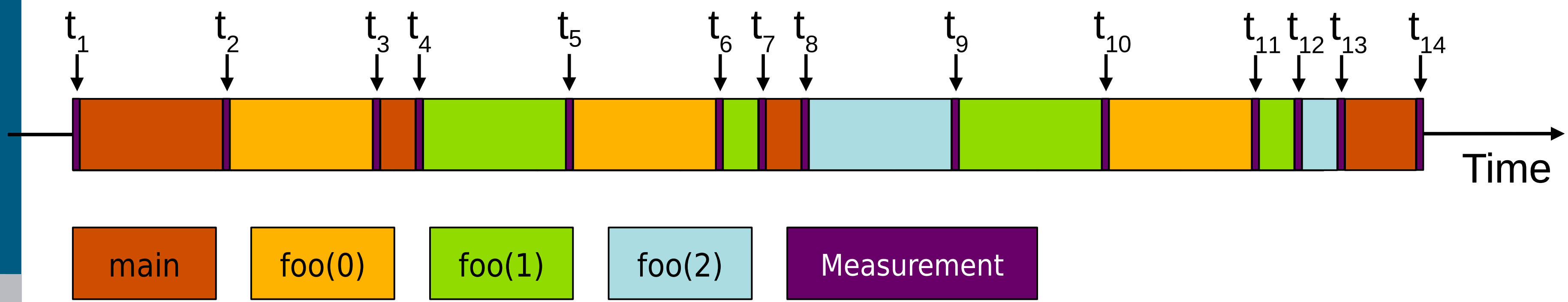
Sampling




```
int main()
{
  int i;

  for (i=0; i < 3; i++)
    foo(i);

  return 0;
}

void foo(int i)
{

  if (i > 0)
    foo(i – 1);

}
```

- Running program is periodically interrupted to take measurement
  - Timer interrupt, OS signal, or HWC overflow
  - Service routine examines return-address stack
  - Addresses are mapped to routines using symbol table information
- **Statistical** inference of program behavior
  - Not very detailed information on highly volatile metrics
  - Requires long-running applications
- Works with unmodified executables

Instrumentation




```
int main()
{
  int i;
  Enter("main");
  for (i=0; i < 3; i++)
    foo(i);
  Leave("main");
  return 0;
}

void foo(int i)
{
  Enter("foo");
  if (i > 0)
    foo(i – 1);
  Leave("foo");
}
```

- Measurement code is inserted such that every event of interest is captured **directly**
  - Can be done in various ways
- Advantage:
  - Much more detailed information
- Disadvantage:
  - Processing of source-code / executable necessary
  - Large relative overheads for small functions

Instrumentation techniques

- Static instrumentation
  - Program is instrumented prior to execution
- Dynamic instrumentation
  - Program is instrumented at runtime

- Code is inserted
  - Manually
  - Automatically
    - By a preprocessor / source-to-source translation tool
    - By a compiler
    - By linking against a pre-instrumented library / runtime system
    - By binary-rewrite / dynamic instrumentation tool

Critical issues

- Accuracy
  - Intrusion overhead
    - Measurement itself needs time and thus lowers performance
  - Perturbation
    - Measurement alters program behaviour
    - E.g., memory access pattern
  - Accuracy of timers & counters
- Granularity
  - How many measurements?
  - How much information / processing during each measurement?

✦ *Tradeoff: Accuracy vs. Expressiveness of data*

Profiling / Runtime summarization

- Recording of aggregated information
  - Total, maximum, minimum, …
- For measurements
  - Time
  - Counts
    - Function calls
    - Bytes transferred
    - Hardware counters
- Over program and system entities
  - Functions, call sites, basic blocks, loops, …
  - Processes, threads

✦ *Profile = summarization of events over execution interval*

Tracing

- Recording information about significant points (events) during execution of the program
  - Enter / leave of a region (function, loop, …)
  - Send / receive a message, …
- Save information in event record
  - Timestamp, location, event type
  - Plus event-specific information (e.g., communicator, sender / receiver, …)
- Abstract execution model on level of defined events

✦ *Event trace = Chronologically ordered sequence of event records*

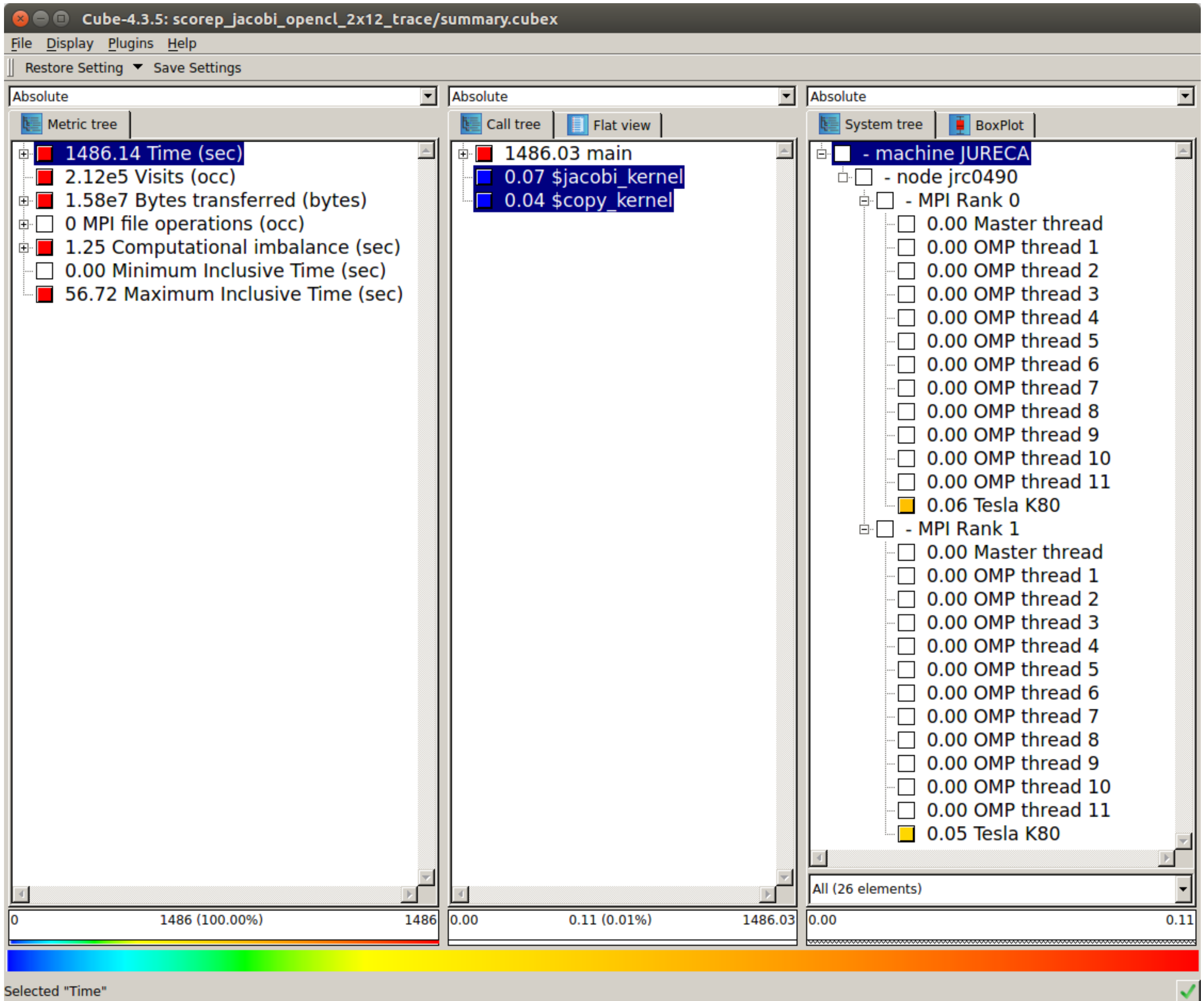
Tracing vs. Profiling

- Tracing advantages
  - Event traces preserve the **temporal** and **spatial** relationships among individual events (✦ context)
  - Allows reconstruction of **dynamic** application behaviour on any required level of abstraction
  - Most general measurement technique
    - Profile data can be reconstructed from event traces
- Disadvantages
  - Traces can very quickly become extremely large
  - Writing events to file at runtime causes perturbation
  - Writing tracing software is complicated
    - Event buffering, clock synchronization, ...

Score-P

- Instrumentation & measurement infrastructure
  - Developed by a consortium of performance tool groups
  - Latest generation measurement system of
    - Scalasca 2.x
    - Vampir
    - TAU
    - Periscope
- Common data formats improve tool interoperability
- http://www.score-p.org/

CUBE4 Interface



Vampir

- Visualization of dynamics
  of complex parallel processes
- Requires two components
  - Monitor/Collector (Score-P)
  - Charts/Browser (Vampir)

Typical questions that Vampir helps to answer:

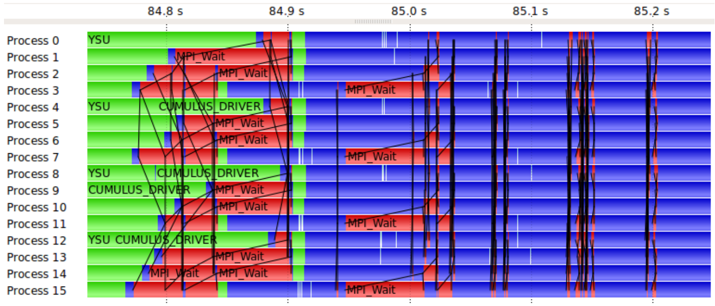
  - What happens in my application execution during a given time in a given process or thread?
  - How do the communication patterns of my application execute on a real system?
  - Are there any imbalances in computation, I/O or memory usage and how do they affect the parallel execution of my application?

Event Trace Visualization with Vampir

- Alternative and supplement to automatic analysis
- Show dynamic run-time behavior graphically at any level of detail
- Provide statistics and performance metrics

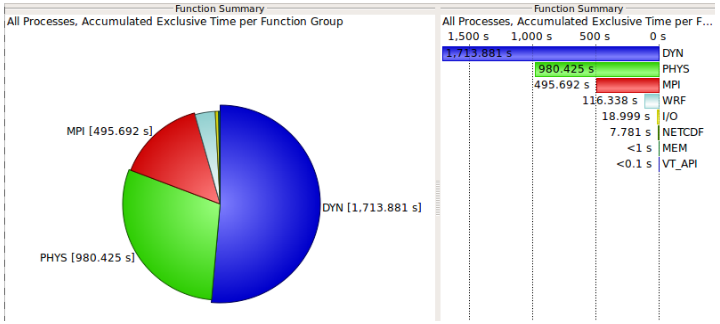
Timeline charts

- Show application activities and communication along a time axis

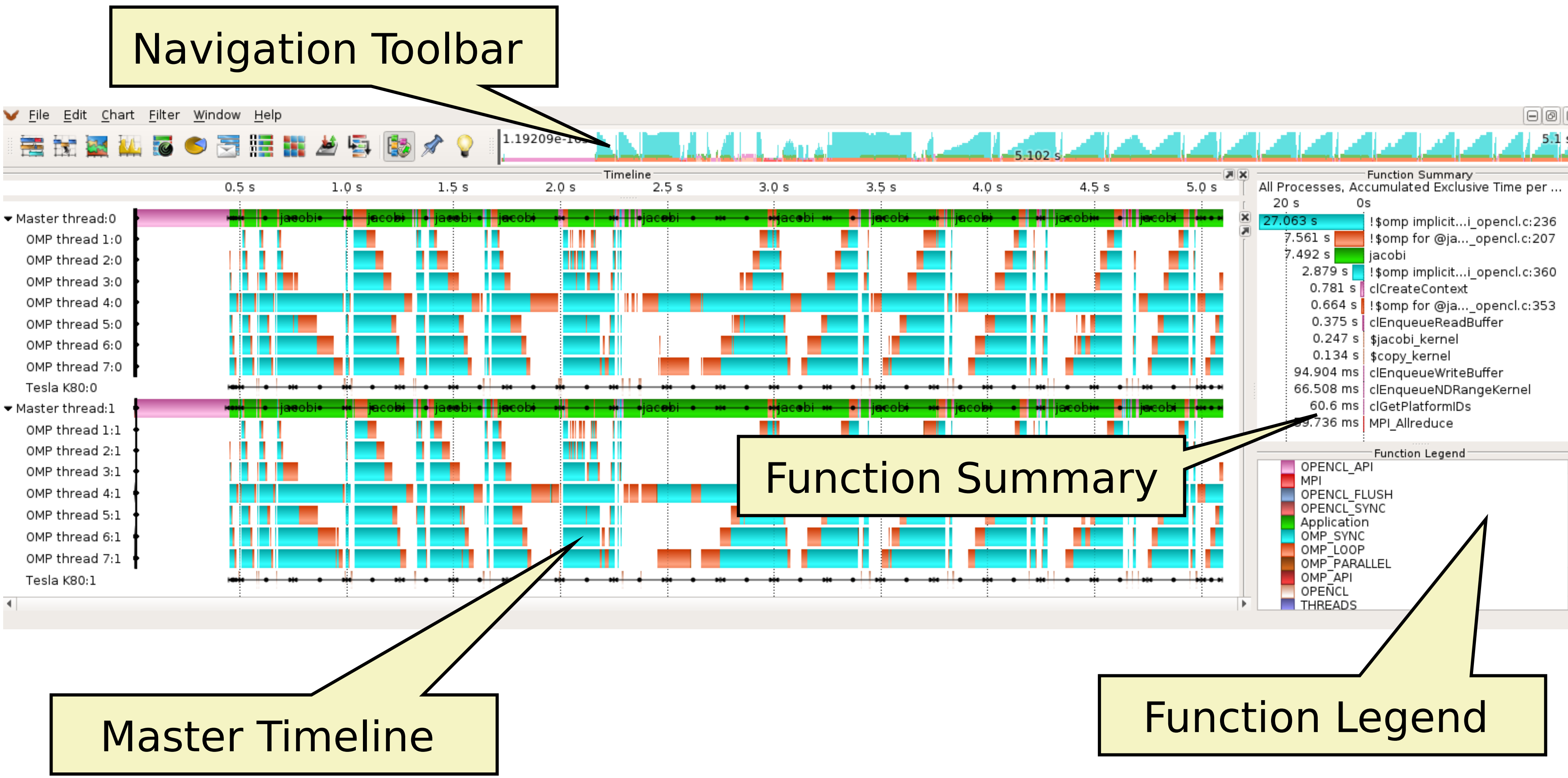


Summary charts

- Provide quantitative results for the currently selected time interval

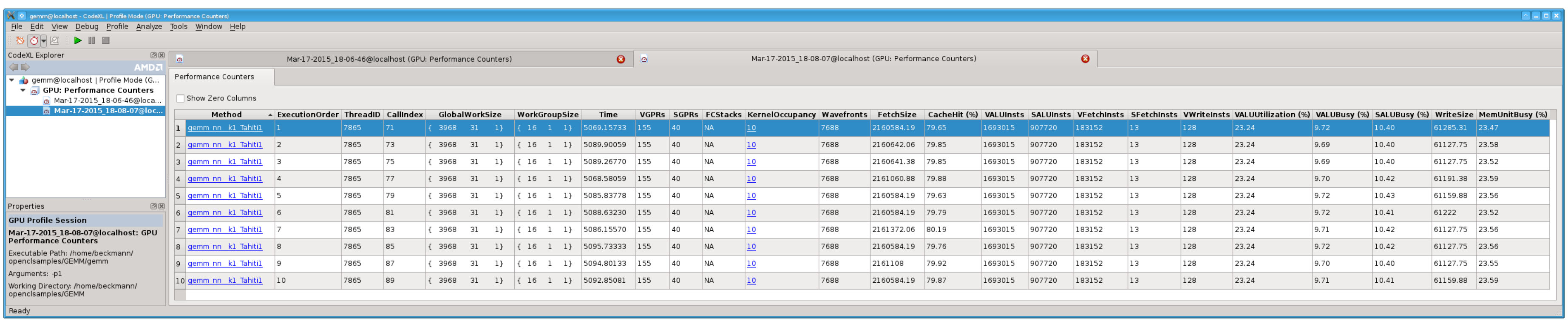


Vampir: Main Interface



AMD CodeXL

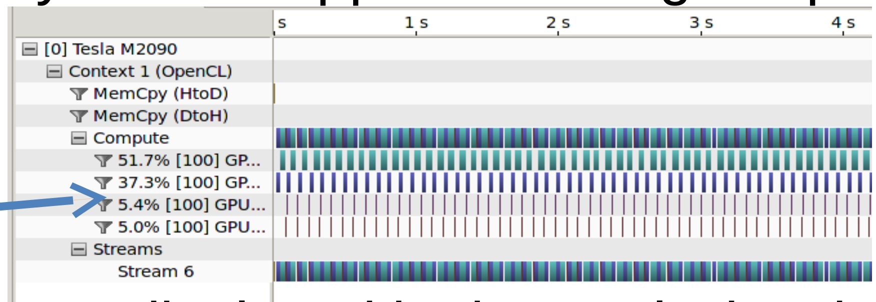
- AMD provide a graphical Profiler and Debugger for AMD Radeon™ GPUs
- Can give information on:
  - API and kernel timings
  - Memory transfer information
  - Register use
  - Local memory use
  - Wavefront usage
  - Hints at limiting performance factors



Profiling using nvvp

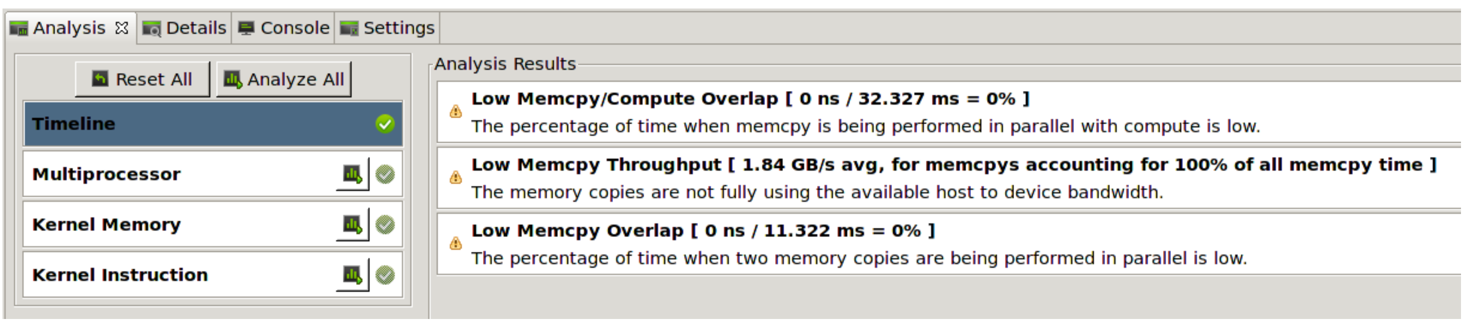
- The timeline says what happened during the program execution:

Kernels



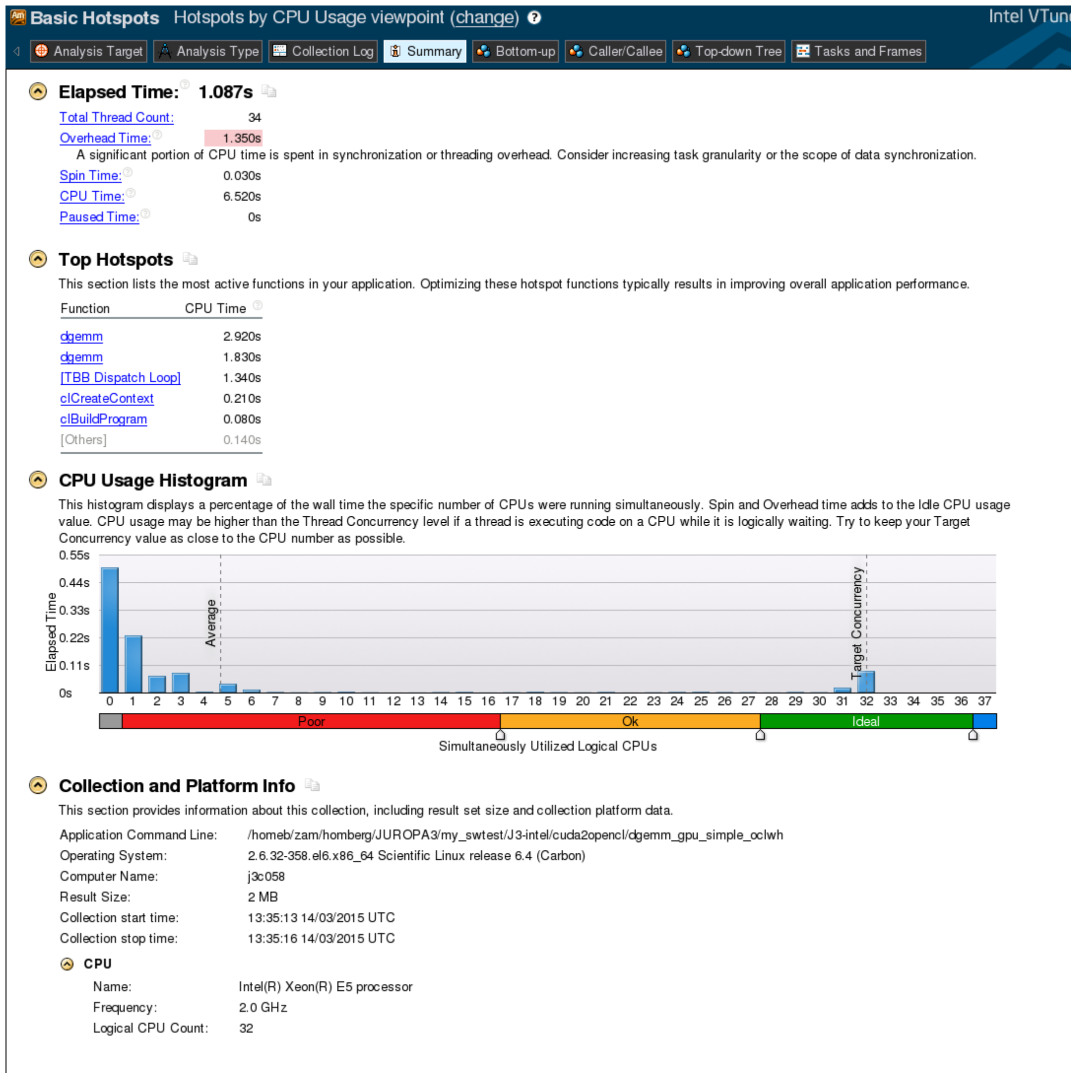
Each invocation of the kernel is pictured as a box

- optimizing tips are displayed in the Analysis tab:



- The Details tab shows information for each kernel invocation and memory copy
  - number of registers used
  - work group sizes
  - memory throughput
  - amount of memory transferred

Intel VTune – Performance analysis tool



Intel Advisor

Program metrics

Elapsed Time	86,10s		
Vector Instruction Set	AVX2, AVX512	Number of CPU Threads	1
Total GFLOP Count	8,12	Total GFLOPS	0,09
Total Arithmetic Intensity	0,07		

Loop metrics

Total CPU time	85,78s	100,0%
Time in **10** vectorized loops	0,78s	
Time in scalar code	85,00s	99,1%

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	6,35x	44%
Program Approximate Gain	1,05x	

Top time-consuming loops

Loop	Self Time	Total Time	Trip Counts
[loop in RichardsJacobianEval at richards_jacobian_eval.c:612]	4,276s	14,792s	2
[loop in RichardsJacobianEval at richards_jacobian_eval.c:612]	2,779s	20,869s	92257
[loop in PhaseRelPerm at problem_phase_rel_perm.c:1102]	1,438s	6,715s	2
[loop in Saturation at problem_saturation.c:280]	1,362s	4,563s	2
[loop in PhaseRelPerm at problem_phase_rel_perm.c:1206]	1,320s	8,897s	2

Recommendations

Loop	Self Time	Recommendations
[loop in PFVDiv at vector_utilities.c:391]	0,140s	Disable unrolling
[loop in NlFunctionEval at nl_function_eval.c:465]	0,080s	Disable unrolling
[loop in PFVScaleBy at vector_utilities.c:1954]	0,020s	Disable unrolling Align data

Tools overview

- Score-P
  - Measurement system to collect profiles and traces
  - http://score-p.org

- CUBE
  - Profile browser
  - http://scalasca.org

- Vampir
  - Trace visualizer
  - https://www.vampir.eu/

- AMD® CodeXL
  - Graphical Profiler and Debugger for AMD® APUs, CPUs and GPUs
  - http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/

- NVIDIA® Nsight™ Development Platform
  - Profiler and Debugger (nvvp) for NVIDIA® GPUs
  - **used to work for OpenCL until CUDA 4.2**
  - https://developer.nvidia.com/nvidia-nsight-visual-studio-edition
  - How to use with OpenCL: http://uob-hpc.github.io/2015/05/27/nvvp-import-opencl/

- Intel Vtune
  - Xeon Phi performance analysis tool
  - https://software.intel.com/en-us/intel-vtune-amplifier-xe

- Intel Advisor
  - Vectorization optimization tool
  - https://software.intel.com/en-us/intel-advisor-xe

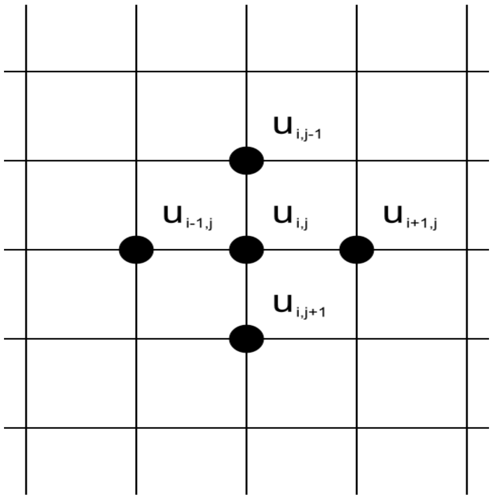
Questions? Ask!

- JSC support
  sc@fz-juelich.de

- VI-HPS tuning workshops
  http://www.vi-hps.org/training/tw

- Apply for free POP performance audit
  https://pop-coe.eu/request-service-form

Hands-on: Jacobi Solver



- Jacobi Example
  - Iterative solver for system of equations

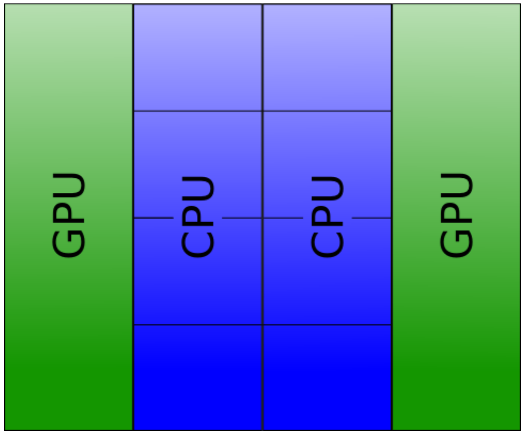
$$U_{old}=U$$

$$u_{i,j}=b\,u_{old,i,j}+a_x\left(u_{old,i-1,j}+u_{old,i+1,j}\right)+a_y\left(u_{old,i,j-1}+u_{old,i,j+1}\right)-rHs/b$$

  - Code uses OpenMP, OpenCL and MPI for parallelization
- Domain decomposition
  - Halo exchange at boundaries:
    - Via MPI between processes
    - Via OpenCL between hosts and accelerators

MPI Process 1

MPI Process 2



Hands-On Exercise

- Copy exercise directory to your working directory on JURECA

```
scp /home/train060/OpenCL_Course/Exercise/jacobi.tar.gz jureca:~
```

- Log on to JURECA

```
ssh -X jureca
```

- Untar jacobi.tar.gz

```
tar xvf jacobi.tar.gz
```

- Load modules

```
module use /homeb/zam/izhukov/modules
module load intel-para CUDA scalasca-ipmpi-cuda
```

- Instrument application

```
PREP="scorep --static" make
```

- Submit batch script

```
sbatch run_scorep.sh
```

Hands-On Exercise

- Or interactively

```
salloc --nodes=1 --time=00:30:00 --partition=gpus --gres=gpu:2 --reservation=OpenCL
OMP_NUM_THREADS=8 srun -n 2 ./bin/jacobi_mpi+opencl <matrix_size_x> <matrix_size_y> <CPU_load>
```

where CPU_load=(0;1)

- Load CUBE and Vampir

```
module load intel-para Cube Vampir
```

- Examine results
  - CUBE

```
cube scorep_jacobi_opencl_sum/profile.cubex
square scorep_jacobi_opencl_sum
```

  - Vampir

```
vampir scorep_jacobi_opencl_trace/traces.otf2
```