Third Party Tools for Titan

Richard Graham, Oscar Hernandez, Christos Kartsaklis,
Joshua Ladd and Jens Domke
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, United States
Email: {rlgraham,oscar,kartsaklisc,laddjs,domkej}@ornl.gov

Jean-Charles Vasnier, Stephane Bihan
and Georges-Emmanuel Moulard
CAPS Enterprise
Rennes, France
Email: {jean-charles.vasnier,stephane.bihan,
georges-emmanuel.moulard}@caps-entreprise.com

***Abstract*—Over the past few years, as part of the Oak Ridge Leadership Class Facility project (OLCF-3), Oak Ridge National Laboratory (ORNL) has been engaged with several third party tools vendors with the aim of enhancing the tool offerings for ORNLs GPU-based platform, Titan. This effort has resulted in enhancements to CAPS' HMPP compiler, Allinea's DDT debugger, and the Vampir suite of performance analysis tools from the Technische Universität Dresden. In this paper we will discuss the latest enhancements to these tools, and their impact on applications as ORNL readies Titan for full-scale production as a GPU based heterogeneous system.**

***Keywords*-Programming Environments; GPU compilers, Performance Tools, Debuggers, GPU programming**

I. Introduction

A Programming Environment (PE) is defined as the software stack that supports the application development cycle for one or more programming models. A typical PE consists of compilers, programming languages, libraries, debuggers, and performance tools unified within a common infrastructure.

During the design of ORNL's OLCF-3 Cray GPU-based system, Titan, ORNL tool developers encountered several challenges. First, there was no production-ready PE for rapidly porting codes to a hybrid GPU-based system that could meet petascale-level performance. Second, it was not clear what the right programming model was and what tools would be necessary to support a massive, hybrid architecture like Titan.

Last year we reported on how we approached the problem [8]. We identified the need for improvements at four ends: compilers, libraries, performance tools and debuggers. For this reason we started working with several vendors, namely CAPS[1], Allinea[2] and Technische Universität Dresden[3], as well as on our research tools HERCULES [9] and Klonos [7], towards our PE requirements. This led to considerable improvements against HMPP [3], DDT [1] and Vampir, such as numerous extensions to HMPP, the addition of HMPP-supporting tools, CUDA-awareness built into Vampir, further DDT scalability, DDT support for HMPP and CUDA, etc.

In this paper we discuss the enhancements made to Titan's PE to support the directives-based hybrid MPI/OpenMP/Accelerator programming model. In this paper we focus specifically on how we enhanced a GPU directives API, as part of the HMPP compiler, to support C++ codes as well as a HMPP runtime API. As part of this paper we describe how these enhancements have been used in the applications. In addition, we describe HMPP extensions which have been developed to manage the coordination of work between CPUs and GPUs and/or multiple accelerators within the node. This will be describe in the context of the LAMMPS [12] application and how it is being ported to a hybrid architecture.

On the debuggers front, we will discuss Allinea's DDT enhancements to fully support GPUs including support for in-kernel CAPS HMPP debugging, fast merging of CUDA threads, and multi-warp stepping. Of particular interest is the collaboration that is taking place between multiple tool vendors allowing different tool sets to work seamlessly with each other. A case study, which will be expanded upon, is DDT's support of HMPP compiler directives. The symbiotic relationship between CAPS and Allinea has resulted in DDT's ability to automatically detect HMPP fragments and allow a user to step into a GPU codelet that is being executed on multiple nodes at scale.

Furthermore, we will discuss the enhancements made within the Vampir toolset to support code development and performance analysis on Titan. These enhancements include, among other things, improved scalability of the Vampir analysis tools, revision of the I/O behavior to reduce file system congestion, and support for CUDA's performance counters via the CUPTI [2] interface. In addition, various enhancements to Vampir's visualization framework, which enable large-scale visualization of CPU/GPU traces, will be described. These include new filter capabilities, support for derived performance counters as well as a display to compare multiple traces and their respective statistics.

[1]http://www.caps-entreprise.com
[2]http://www.allinea.com/
[3]http://tu-dresden.de

II. HMPP Overview

CAPS HMPP is one of the third-party compilers in Titan. It is a directives-based compiler and is capable of generating kernels that have been accelerated in a hybrid CPU/GPU manner, as well as executing them via the HMPP runtime. HMPP directives can be added incrementally to existing applications and are compatible with OpenMP and MPI. One benefit of using directives is that they preserve legacy code by discouraging programmers from targeting specific hardware and thus directives facilitate maintenance beyond the lifetime of a given platform. This is showcased by HMPP which can generate code for various GPU architectures or accelerators.

HMPP is a source-to-source compiler that uses standard compilers (GNU gcc/gfortran, Intel icc/ifort and PGI pgcc/pgfortran compilers) as backends to produce the host binary code. It integrates a C/Fortran-to-CUDA code generator that produces accelerated versions of kernels as variants of computations. CUDA code is compiled as shared libraries using the NVIDIA compiler. HMPP-built applications are linked with the HMPP runtime that loads and executes the CUDA code if the GPU is present, or runs the CPU version otherwise. HMPP directives are safe meta-information added to the application source code that preserves the semantics of the the original code. They address the remote execution (RPC) of functions or regions of code as well as the transfers of data to and from the accelerator memory. In addition to HMPP-specific directives, the HMPP compiler also supports the new OpenACC directive-based parallel programming standard, which was initially developed by PGI, CRAY and NVIDIA.

A. C++ Support

HMPP version 3 was extended to support the acceleration of C++ codes. HMPP provides a GPU directive and a C++ class API to define codelets that will run in the GPU. Both mechanisms are used to coordinate the work among CPU and GPUs. The use of the directive is illustrated in the following example:

```
template<typename T>
void add_vector(int n, T * y, T * a, T * b)
{
  for(int i = 0 ; i < n ; i++)
      y[i] = a[i] + b[i];
}
// Instanciate the add_vector<float> template
// and associate it to the codelet "add_vector_float"

#pragma hmppcg entrypoint as add_vector_float,target=CUDA
template void add_vector<float>(int n,
float *y, float *a, float *b);
```

Figure 1. The HMPP C++ directive that used to define a method that is translated to CUDA or accelerator code

The directive *#pragma hmppcg entrypoint* defines a GPU codelet in a C++ class. Figure 1 shows how this directive can be applied to a C++ template. When the code is compiled, the HMPP C++ compiler will instantiate the template and automatically generate the CUDA version of the code for the codelet *add_vector_float*. The user need to allocate a GPU device (where the codelet will run) and define two C++ objects, one for each of the classes *Grouplet* and *Codelet*, to reference the *add_vector_float* generated kernel. The HMPP C++ API provides the classes and methods to do this. Figure 2 shows how the HMPP C++ API is used to define an object of class *device* for allocating a GPU device and how to retrieve the codelet *add_vector_float* from a grouplet.

```
// Get an interface to the first available
// CUDA compatible device
hmpprt::Device *device;

device = hmpprt::DeviceManager::getInstance()->
                                  acquireDevice(hmpprt::CUDA);
// Dynamically loads a grouplet which has been generated
// by from current file and the CUDA target

hmpprt::Grouplet *grouplet;
grouplet = new hmpprt::Grouplet(__FILE__, hmpprt::CUDA);

// Get the handle of the "add_vector_float"
// generated codelet
hmpprt::Codelet *codelet;
codelet = grouplet->getCodeletByName("add_vector_float");
```

Figure 2. Code snippet that shows how to define a grouplet, codelet and an accelerator device using the HMPP C++ API

Figure 3 shows a code snippet where the data of the codelet parameters are allocated and uploaded to the GPU memory using the HMPP C++ API. In HMPP for C++, the user needs to create a codelet argument list object that is

```
const int nbValues = 16;
vector<float> Y;
vector<float> A;
vector<float> B;

// Allocate the data for each parameter
hmpprt::Data *dataA;
dataA = new hmpprt::Data(device, nbValues * sizeof(float),
codelet->getMemorySpaceByName("a"));

hmpprt::Data *dataB;
dataB = new hmpprt::Data(device, nbValues * sizeof(float),
codelet->getMemorySpaceByName("b"));

hmpprt::Data *dataY;
dataY = new hmpprt::Data(device, nbValues * sizeof(float),
 codelet->getMemorySpaceByName("y"));

dataA->allocate();
dataB->allocate();
dataY->allocate();

// Upload data from the host memory to the device memory
dataA->upload(&A[0]);
dataB->upload(&B[0]);
```

Figure 3. A code snippet that shows how data is allocated and uploaded to GPU memory using the HMPP C++ API

used to invoke the codelet with the method *call* from the device object. Figure 4 shows how this is done.

```
// Create the codelet argument list
hmpprt::ArgumentList arguments;

// Call the codelet
arguments.addArgument(Y.size());
arguments.addArgument(dataY);
arguments.addArgument(dataA);
arguments.addArgument(dataB);
device->call(codelet, arguments);
```

Figure 4. Code snippet that shows how invoke a codelet using the HMPP C++ API

Finally the result can be transferred from the GPU to the host (CPU) and the GPU memory is freed. This is done by invoking the download and free methods from the HMPP data object. This is shown in Figure 5.

```
// Download output data from the device to the host
dataY->download(&Y[0]);

dataA->free();
dataB->free();
dataY->free();
```

Figure 5. Code snippet that shows how to transfer data from the GPU to the host and free the GPU memory

B. Data Distribution Support

As the number of cores keeps growing, it is essential to help developers to easily distribute data and computations over multiple CPUs and GPUs. The HMPP directives support CPU and multi-GPU programming by enabling developers to spread out a collection of data on multiple devices and the CPU. This feature was developed with the idea to support the LAMMPS application using the C++ API of HMPP. The HMPP data distribution directives can also be used in C and Fortran code. Figure 6 shows the data distribution directive that can be placed before a parallel code region such as a loop.

```
//expr : a device number
#pragma hmpp <myGroup> parallel, device="expr"
```

Figure 6. The HMPP data distribution directive

Each directive in the parallel region is pushed in a queue. All directive operations of the queue are then executed over the devices at the end of the parallel region. The directive operations in the parallel region are associated to a device whose number is defined according to the value of the device clause expression. For instance, if the expression of the device clause is based on the loop induction variable, each directive operation will be executed in parallel in a device whose number depends on the iteration value. Figure **??** shows a data distribution code example, where the distribution of a collection of data is performed over two CUDA devices.

```
#pragma hmpp <my_grp>  parallel, device="i%2"
  for (i = 0; i < N; ++i)
  {
#pragma hmpp <my_grp> new, data["x[i]"]
#pragma hmpp <my_grp> allocate,data["x[i]"], &
data["x[i]"].size={10000}

#pragma hmpp <my_grp> my_cdlt callsite
    f(x[i]);
#pragma hmpp <my_grp> release, data["x[i]"]
#pragma hmpp <my_grp> delete, data["x[i]"]
  }
```

Figure 7. The HMPP data distribution directive is used to distribute the array x on multiple devices.

In the case of LAMMPS, the same effect was achieved using the HMPP C++ API to distribute half of the data between the GPU and CPU for the *computeCodelet* kernel. Figure II-B shows the allocation of the two devices, a CPU and a GPU device, using the HMPP C++ API.

```
//allocation of the devices: GPU as device 0
//and CPU as device 1
devices[0] = hmpprt::DeviceManager::getInstance()->
getFirstCUDADevice();
devices[1] = hmpprt::DeviceManager::getInstance()->
getHostDevice();

//retrieve the grouplets using the
// name of the generated .so
grouplets[0] = new hmpprt::Grouplet("pair_lj_cut-cuda.so");
grouplets[1] = new hmpprt::Grouplet("pair_lj_cut-host.so");

// retrieve the codelets
codelets[0] = grouplets[0]->
getCodeletByName("computeCodelet");
    codelets[1] = grouplets[1]->
    getCodeletByName("computeCodelet");
  }
}
```

Figure 8. The HMPP C++ API to allocate two devices (a GPU and a CPU) that will be used to distribute the data elements of the array f for the *computeCodelet* codelet

The array f of the computeCodelet function is split in two different arrays associated to two different devices: a CPU and a GPU. All other arrays of the function are fully allocated on each device. The effects of the data distribution is shown in Figure 9.

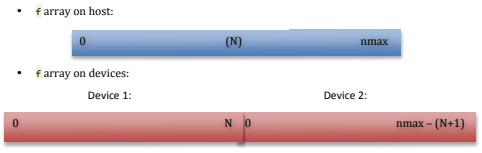

Figure 9. The distribution of the array f between a GPU and CPU device for the LAMMPS application

Figure 10 shows the allocation and upload of the distributed array for the two devices:

```
 //size of first part of f
int size0_f = (inum / nb_devices ) * 3;
//size of rest of f
int size1_f = (atom->nmax-(inum / nb_devices)  ) *3)

//allocate first part of f on device 0
data_f[0] = new hmpprt::Data(devices[0],
sizeof(double) * size0_f);

//allocate rest of f on device 1
data_f[1] = new hmpprt::Data(devices[1],
sizeof(double) * size1_f);

//send first part of f on device 0
data_f[0]->upload(f[0]);

//send the rest of f on device 1
// f[inum /nb_devices]): address of
data_f[1]->upload(f[inum /nb_devices]);

// the beginning of the second part
    ...
```

Figure 10. The allocation and upload of the distributed array on two devices using the HMPP C++ API

Finally the codelet is executed in the GPU and CPU using a loop. This is shown in Figure 11.

```
//execution of the compute function with f array
//distributed on two devices
for(int i=0; i<nb_devices; i++){
  //creation of the list of arguments
  arg_list[i] = new hmpprt::ArgumentList(codelets[i]);

  //add the codelet arguments to the list
  arg_list[i]->addArgument(atom->ntypes + 1);
  arg_list[i]->addArgument(atom->nlocal);
  arg_list[i]->addArgument(nlocal + atom->nghost);
  arg_list[i]->addArgument(split[i *2 + 0]);
  arg_list[i]->addArgument(split[i *2 + 1]);
  arg_list[i]->addArgument(force->newton_pair);
  arg_list[i]->addArgument(data_x[i]);
  arg_list[i]->addArgument(data_f[i]);
  arg_list[i]->addArgument(data_type[i]);
  arg_list[i]->addArgument(data_numneigh2[i]);
  arg_list[i]->addArgument(data_special_lj[i]);
  arg_list[i]->addArgument(data_cutsq[i]);
  arg_list[i]->addArgument(data_lj1[i]);
  arg_list[i]->addArgument(data_lj2[i]);
  arg_list[i]->addArgument(data_ilist[i]);
  arg_list[i]->addArgument(data_full_j[i]);

  //execute the codelet in the two devices
  devices[i]->call(codelets[i],*(arg_list[i]));
}
```

Figure 11. A code snippet that shows the invocation of the *computeCodelet* that will be executed in the GPU and CPU where the data has been distributed between two devices

C. Library Support

The LSMS [13] application calculates the interactions between electrons and atoms in magnetic materials. The CPU profile of the application showed that the BLAS and LAPACK functions take about 95% of the execution time. Figure 12 shows the part of the code where calls to these libraries are done.

Figure 13 shows how the LSMS library calls are ported to GPU by replacing calls to BLAS and LAPACK by calls to NVIDIA CUBLAS and CULA.

```
do iblk=nblk,2,-1
    call zgetrf(...)
    call zgetrs(...)
    call zgemm(...)
    call zgemm(...)
enddo
call zgemm(...)
```

Figure 12. LSMS calls to the BLAS and LAPACK libraries

```
do iblk=nblk,2,-1
    ...
    call cula_device_zgetrf(...)
    call cula_device_zgetrs(...)
    call cublas_zgemm(...)
    call cublas_zgemm(...)
enddo
call cublas_zgemm(...)
call cublas_get_matrix(...)
```

Figure 13. LSMS calls to the BLAS and LAPACK libraries are replaced by calls to NVIDIA CUBLAS and CULA

As there is no one-to-one mapping among similar functions in libraries like BLAS and cuBLAS, calls to a CPU library call cannot be directly replaced by a call to the GPU version of the library. HMPP offers a proxy mechanism to seamlessly deal with this. HMPP allows the user to replace the calls of original library functions by their equivalent GPU version (using directives) while keeping the original CPU library calls in the code. A hmppalt directive with the name of the proxy is inserted before the calls to the original library functions (See lines 18, 20, 22, 24, 27 in Figure 14).

Depending on the execution context, the proxy can call either be the CPU or GPU version of the library. Each library function used in the proxy needs to be declared in the application source file with the hmppalt declare directive (see lines 1, 6 and 11 of Figure 14). At compile time, HMPP generates two versions of the code, the first one calling the GPU library function in the proxy and the second one using the original library. A code snipped of the HMPP proxy definition is shown in Figure 15.

Figure 15 shows that when the proxy executes the GPU version of the library, it gets the mirror of the data in the device memory and only calls the CUBLAS zgemm function. Having a proxy simplifies the process of calling different versions of the library while achieving the same performance as the original or the GPU version of the code.

III. DDT Debugger

A scalable, hybrid aware debugger is a critical component of Titan's programming environment. By employing sophisticated tree topologies, Allinea, working alongside ORNLs Applications Performance Tools, have deployed the DDT debugger. Field-tested on actual development code at ORNL, DDT has been shown to scale-up to over 200,000

```
1. !$hmppalt myproxy declare, name="zgemm" ,
          extend(error,...), fallback=true
2. SUBROUTINE lsmszgemm(error,transa,transb,
          m,n,k,alpha,a,lda,b,ldb,beta,c,ldc)
3. ...
4. END SUBROUTINE lsmszgemm
5.
6. !$hmppalt lsms declare, name="zgetrf" ,
          extend(error,...), fallback=true
7. SUBROUTINE lsmszgetrf(error, m, n, a,
          lda, ipiv, info)
8. ...
9. END SUBROUTINE lsmszgetrf
10.
11. !$hmppalt lsms declare, name="zgetrs" ,
          extend(error,...), fallback=true
12. SUBROUTINE lsmszgetrs(error, trans, n, nrhs, a,
          lda, ipiv, b, ldb, info )
13. ...
14. END SUBROUTINE lsmszgetrs
15. ...
16. !$hmpp advancedload
17. do iblk=nblk,2,-1
18. !$hmppalt myproxy
19. call zgetrf(...)
20. !$hmppalt myproxy
21. call zgetrs(...)
22. !$hmppalt myproxy
23. call zgemm(...)
24. !$hmppalt myproxy
25. call zgemm(...)
26. enddo
27. !$hmppalt myproxy
28. call zgemm(...)
29. !$hmpp delegatedstore
```

Figure 14. A hmppalt proxy directive is inserted before the calls to the original library functions. This will enable the HMPP to call either the CPU or GPU version of the library.

```
void lsmszgemm_(){

hmpp_mirror = hmpprti_get_mirror(mirror);
 A = hmpprt_data_get_device_address(hmpp_mirror);


hmpprt_cuda_lock_context();
cublaZgemm();
hmpprt_cuda_unlock_context();

}
```

Figure 15. A code snippet of the HMPP proxy.

cores. Allinea has employed a co-design methodology in developing DDT, working closely with Cray and CAPS to tightly integrate DDT into Crays advanced programming environment. As a result, DDT is a debugger designed from the ground-up for large-scale hybrid systems. In addition to MPI/OpenMP parallel programs, DDT is fully supported on NVIDA GPUs and is capable of stepping into CUDA kernels with multi-warp stepping capabilities. In addition, DDT is capable of stepping into and through HMPP codelets. Recently, Allinea has added support for CUDA memory debugging capabilities, allowing a user to visualize the physical memory layout of their code spread over both host and device memory.

IV. VAMPIR/VAMPIRTRACE PERFORMANCE TOOL

One performance analysis tool for parallel and hybrid applications is the Vampir toolset [6], [10]. It consists of three major components, named the Open Trace Format library (OTF), VampirTrace and Vampir. VampirTrace is used in the pre-run phase to prepare the source code or binary of the application with instrumentation of function calls, library wrapping, etc, to gather events during the run. At runtime VampirTrace, i.e. libraries linked into the binary, processes the events and passes them to the OTF library. OTF will then save the events, depending on their origin, into trace streams and will write one file containing the event stream for each MPI process, OpenMP thread and thread executed on an accelerator. Besides that, OTF provides post-processing tools to de-/compress, modify and analyze the trace. The visualization and analysis component of the Vampir toolset is the Vampir client and server, henceforth referred to as Vampir. The purpose of the server is to analyze the trace in parallel and aggregate enough main memory on the compute nodes to open/load the trace. The client visualizes the data analyzed and transferred by the server, and offers a variety of methods to interact with the trace, like scrolling, zooming, highlighting of interesting areas, etc.

A. Enhancements for OLCF-3

The next step in terms of scalability of the OLCF-3 project brought new challenges for the performance analysis tools. In the beginning of OLCF-3, we identified four major areas where the Vampir toolset needs to be improved to allow the performance analysis of the whole Titan system. The first and obvious challenge is the support for tracing of GPUs. Besides that, the toolset had to be enhanced in the scope of I/O performance, scalability and usability. In the following, we will introduce the changed made by the vendor up to now.

In cooperation with NVIDIA, the vendor was able to support always the latest CUDA library version, which is version 4.1 right now, inclusive CUPTI 2.0 GPU performance counters. To fit into the general design of processes/threads and messages within Vampir, the CUDA threads will be visualized as threads, which belong to one process, and the memory transfers between host CPU and GPU are shown as black lines, which is similar to MPI messages. VampirTrace is capable of tracing multiple/different CUDA kernels executed by one binary and the developers improved the accuracy of the timing information for asynchronous kernel execution while using CUPTI to measure the timing-events. A visualization of the result is shown in Fig. 16. The traced program is a GPU-accelerated Monte Carlo simulation for x-ray transport[4].

In terms of scalability, the vendor implemented MPI parallelized versions of their post-processing tools, like otfprofile,

[4]http://code.google.com/p/mcgpu/

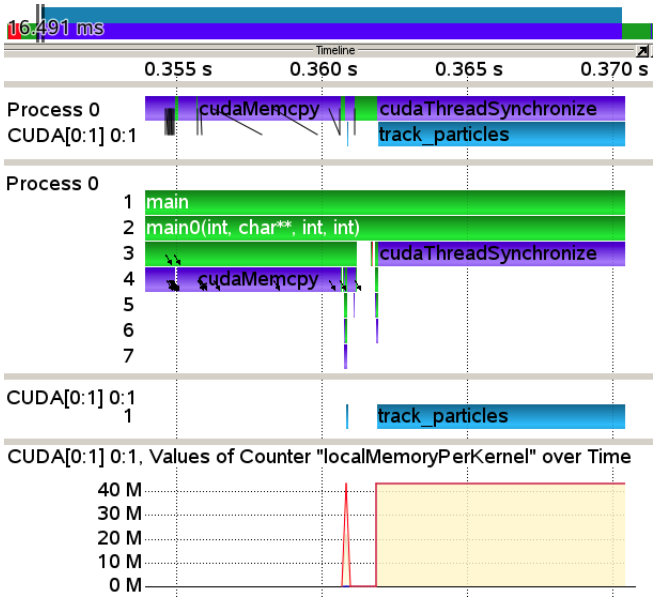

Figure 16. Master timeline, call tree for process 0 and the CUDA thread, and GPU counter "localMemomyPerKernel" for a Monte Carlo simulation on the GPU

otfmerge and vtfilter, to enable the post-processing of traces with more than 10.000 streams. The Vampir developers improved the MPI behavior of vampirserver to run efficiently with more than 10.000 analysis processes, which allows the user to analyze traces with over 200.000 streams, i.e. analyzing a trace of an application utilizing complete Titan is possible.

To relieve to load for the meta data server of Lustre, the VampirTrace/OTF developers implemented routines to use the IOFSL library [5]. In doing so, the user is able to allocate a small number of compute nodes dedicated to run the IOFSL servers, whereby each server aggregates multiple trace streams into one file. A reduction/compression of the trace data during runtime is available within VampirTrace. If VampirTrace identifies patterns inside of the stream, e.g. similar iterations of a do-loop, then it saves one representative. Vampir is capable to duplicate those representatives and visualize/analyze the trace as a whole without additional memory requirement.

Besides the previous discussed improvements, the vendor was working on the usability and enhanced the analysis possibilities within Vampir and VampirTrace. The usability features are among others the accessibility, i.e. predefined colors for humans with daltonism, and the Vampir client was made available for Mac OS X, wherewith the client can be used on all three common operating system: Linux, Mac OS X and Windows. Additionally, the vendor developed the automatic generation of filter files based on previous runs, so that VampirTrace will only collect the performance relevant events of the subsequent run. To get an general overview of the trace data without loading it into with Vampir, the output of otfprofile was enhanced. This includes global message statistics, like minimum, maximum and average message sizes, the time spend in MPI routines, i.e. minimum, maximum and the average time across processes, and process clustering information, which is similar to the process clustering within Vampir. The enhanced analysis possibilities within Vampir include derived counters, which can be defined by the user and can be made up of hardware performance counters. In addition, the client offers new filter capabilities, e.g. filtering of functions, functions groups or processes, and offers the highlighting of function anomalies using thresholds of performance or timing values to assist the user with the analysis and the search for bottlenecks. One problem was, that Vampir had only a display to visualize one performance counter for one stream. To get an overview of one counter for all processes over time, the Performance Radar display was developed, which offers an easier way to identify bottlenecks or computational imbalances inside of the application. Furthermore, a new display was implemented to directly compare two or more traces. This display enables the time-wise alignment of the traces and it enables the visualization of common Vampir displays next to each other, i.e. the user does not has to switch between windows to compare the cache misses for two traces, for instance. Figure 17 shows an example of the "Compare View" display for four different runs of land model simulations within the Community Earth System Model [11]. Based on different input sets for spring, summer, fall and winter, we are able to identify variations in the computational intensity in the traces.

V. Conclusion

The purpose of this paper has been to present the enhancements to the CAPS, DDT and Vampir/VampirTrace tools for the upcoming Cray-based Titan system. We continue to collaborate with a number of vendor to enhance their offerings as part of the Cray PE hardening. The PE is being tested on existing systems and applications already make use of it and follow up with its developments. These tools are available to be used at production level on Titan.

Acknowledgment

This work was funded by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. This research used resources of the Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UTBattelle, LLC.

References

[1] The Distributed Debugging Tool. http://www.allinea.com, 2008.

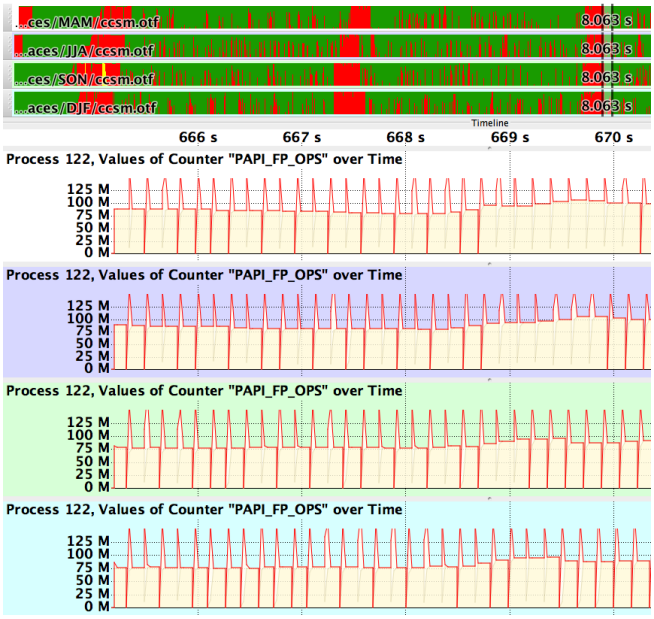

Figure 17. The Compare View for the land model simulation of CESM shows small but important variations in the computational intensity. The computational intensity in the land model is among others influenced by the solar radiation, which differs with the change of seasons.

[2] CUDA Tools SDK: CUPTI User.s Guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUPTI_Users_Guide.pdf, 2011.

[3] HMPP: Many-Core Programming Environment. http://www.caps-entreprise.com/upload/ckfinder/userfiles/files/HMPP_HybridMulticoreParallelProgramming/CAPS_PROD_EN_HMPP.pdf, 2011.

[4] The OpenACC Application Programming Interface. http://www.openacc-standard.org/Downloads/OpenACC.1.0.pdf, 2011.

[5] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable I/O forwarding framework for high-performance computing systems. In *IEEE International Conference on Cluster Computing (Cluster '09)*, pages 1–10, September 2009.

[6] Holger Brunst and Andreas Knüpfer. Vampir. In *Encyclopedia of Parallel Computing*. Springer, October 2011.

[7] Wei Ding, Oscar Hernandez, Chung-Hsing Hsu, Richard Graham, and Barbara M. Chapman. Bioinspired similarity-based planning support for the porting of scientific applications. In *4th Workshop on Parallel Architectures and Bioinspired Algorithms*, Galveston Island, Texas, USA, 2011.

[8] R. Graham, P. Shamis, O. Hernandez, C. Kartsaklis, T. Mintz, and C. Hsu. A Programming Environment for Heterogeneous Multi-Core Computer Systems. In *Cray User Group (CUG)*, May 2011.

[9] C. Kartsaklis, O. Hernandez, C. Hsu, T. Ilsche, W. Joubert, , and R. Graham. HERCULES: A Pattern Driven Code Transformation System. In *International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, May 2012.

[10] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg, 2008.

[11] Keith Oleson, David Lawrence, Gordon Bonan, Mark Flanner, Erik Kluzek, Peter Lawrence, Samuel Levis, Sean Swenson, Peter Thornton, Aiguo Dai, Mark Decker, Robert Dickinson, Johannes Feddema, Colette Heald, Forrest Hoffman, Jean-Francois Lamarque, Natalie Mahowald, Guo-Yue Niu, Taotao Qian, James Randerson, Steve Running, Koichi Sakaguchi, Andrew Slater, Reto Stockli, Aihui Wang, Zong-Liang Yang, Xiaodong Zeng, and Xubin Zeng. Technical Description of version 4.0 of the Community Land Model (CLM). Technical Report NCAR/TN-478+STR, National Center for Atmospheric Research, April 2010. http://nldr.library.ucar.edu/repository/collections/TECH-NOTE-000-000-000-848.

[12] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1 – 19, 1995.

[13] Timothy J. Sheehan, William A. Shelton, Thomas J. Pratt, Philip M. Papadopoulos, Philip LoCascio, and Thomas H. Duni gan. The locally self-consistent multiple scattering code in a geographically distributed linked mpp environment. *Parallel Computing*, 24(12-13):1827 – 1846, 1998.