A Programming Environment for Heterogeneous Multi-Core Computer Systems

ORNL
Richard Graham, Bronson Messer, Oscar Henrandez, Christos Kartsaklis, Thomas Ilsche, Tiffany Mintz, Wayne Joubert, Robert Whitten, Ricky Kendall

May 6, 2011

Abstract

As part of the Oak Ridge Leadership Computing Facility (OLCF)-3 project, the Oak Ridge National Laboratory (ORNL) is working with several vendors and engaged in research to develop a Programming Environment (PE) for mixed CPU/GPU based ultra-scale computer systems. The environment provides a toolset to port or develop CPU/GPU systems while reducing development time to improve the performance and portability of the codes while minimizing sources of errors. Our toolset consists of compilers for high-level GPU directives, libraries, performance tools, and a debugger with synergistic interfaces among them. In this paper we show how these tools work together and how they support the different stages of the program development and porting cycle. We describe how we successfully used the tools to port Department of Energy codes to a CPU/GPU system.

1 Introduction

A PE is defined as the software stack that supports the application development cycle for one or more programming models (PM). The compiler, programming language(s), debuggers and profilers are components of a typical PE.

ORNL's OLCF-3 Cray GPU-based system, Titan [5], poses a number of challenges. There has not been a production-grade PE for production HPC systems to address challenges, such as rapidly porting codes to hybrid GPU-based systems, and meeting petascale-level scalability requirements, and this is what we are establishing. Vendors, and application and tool developers are working hand-in-hand so that OLCF-3 users' productivity can be increased and the porting of their codes simplified as demonstrating the system's performance through user codes is a certain requirement. Consequently, our work comprises the assessment and proposal of enhancements towards the Cray PE, partnerships for readying third-party tools for our Cray-based PE (in terms of requesting extensions for OLCF-3 requirements and future trends), and the direct integration of third-party tools (i.e. "as they are") with the PE.

Specifically, we have collaborated with CAPS to extend its HMPP language with features that OLCF-3 application developers will directly benefit from, and to produce HMPP-aware performance analysis and code-refactoring tools. We have worked with the TU Dresden to make Vampir CUDA-aware, as the CUDA runtime will most likely occur in the software stack and we have collaborated with Allinea to make the Allinea DDT applicable to both the Cray PE and HMPP. Finally, our commitment in the establishment of the PE can also be highlighted by being members of OpenMP Architecture Review Board (ARB).

Figure 1 depicts, in terms of the framed items and the edges among them, what we consider as a representative work-flow that our users engage in, plus the selection of tools that implement it. With regards to the Titan system, we have been working on complementing the Cray PE with a number of third-party tools, some of which are shown in Figure 1. We have partnered with the vendors of these tools for the purpose of extending them so that they meet the requirements of our Cray-based PE.

The remainder of this paper is organized as follows: We begin by discussing the PE at the language and compiler level, then move on to tools that assist with analysis of programs, then profiling and tracing tools of the PE, and finally the debugging elements of the PE. The description is mostly vendor- and application-oriented.

2 Languages and Compilers

An important aspect of a successful PE is to increase users' producitivy with regards to performance engineering and application development processes; processes such as porting are generally regarded as a combination of these two.

Let us firstly outline the basic properties of the PM. The PM will support distributed memory and the heterogeneity of the system will be managed mainly with high level directives. These directives, apart from tasks such as work distribution and data motion, should also ease code restructuring, which is often where most time is spent. We consider a distributed memory model across the nodes (MPI programmable) and shared memory within the nodes (OpenMP programmable). Node devices will support both the shared and distributed model by software-based concurrency control, and shadowing mechanisms (for discrete GPUs). Device memory will also be more prominent in that it will be addressable during inter-node operations (e.g. transfering data to a remote GPU memory) and opportunities such as disjoint heterogeneous caching[1] and scratchpad programming should be exploited. Overall, however, there will be more explicit data motion control for reasons of power efficiency. Both the task-parallel and the data-parallel model will be featured. The hybrid CPU-GPU PM poses challenges at both the data and program-structuring levels.

At the data level, the following are challenges that need to be addressed: (1) data need to be copied in and out of the accelerator device manually (lack of integrated memory) and often not as a whole (worksharing), (2) data needs to be staged in accelerator memories so that their access pattern matches the capabilities of these memories, (3) data often need reorganization if being present in multiple access patterns, (4) data whose production/consumption spans device boundaries often need their consistency maintained, and (5) as data motion is more expensive, latency hiding mechanisms need to be employed. On the other hand, there are

[1]i.e. the presence of separate caches in the same processing unit, each featuring different caching policies.

program-structuring level challenges too, such as: (1) separate host and device code needs to be developed and maintained, (2) accelerators differ greatly at the PM level, (3) the presence of multiple accelerating units (cores, SIMD units, multiple GPUs) must be supported at the work distribution level (at least), and (4) tools that help users with utilizing the language facilities of the PM (e.g. preparatory steps for using particular directives).

Directives (compiler pragmas) are a convenient means for program restructuring and for conveying information to the compiler. They promote incremental porting and development, which leads to rapid emission of code for accelerators, and are recognized by tools, which simplifies the association of sources with tools' analysis. We are evaluating Cray's implementation of OpenMP extensions for accelerators [4] in the Cray CE as well as being members of the OpenMP ARB. The next paragraphs document features that have been developed in HMPP [2] as part of our partnership with CAPS, have reached or are near completion, and which are currently being utilized by OLCF applications. HMPP is a directives-based compiler that can generate parallel code for hybrid platforms and which hides knowledge of the target hardware from the programmer.

Copying data in and out of accelerator devices is a time-consuming process as the data do not always have a flat layout (e.g. an array of primitive data types). HMPP has been extended to support user-defined data types as well as data structures holding pointer field; our applications, such as the Community Atmosphere Model/Spectral Element (CAM/SE) and others rely on user defined data types to store the cubed elements information. With the introduction of dynamic CPU/GPU coherency management our users are relieved from manually mirroring host/device images of data structures upon modification. Requesting coherency maintenance through a directive as opposed to implementing it by hand reduces code size greatly and is type-agnostic.

Users often need to contrast the performance of hand-tunned, compiler-generated and external (e.g. library-provided) kernels. The implementation of User-Kernel Integration instructs HMPP to bypass its own code generation and utilize user-supplied code directly, and thus, it achieves the desired effect. The Locally Self-consistent Multiple Scattering [14] (LSMS) developers are in the process of modifying their application so that it can make use of CULA [1] (a GPU linear algebra library) in terms of this facility. Our partnership with CAPS has also led to the formation of HMPP++. HMPP++ bridges HMPP and OO programming by allowing application C++ classes to inherit from the HMPP runtime's classes while utilizing fully the HMPP directives (extended to by C++ scope-aware, etc.) at the same time; this hybrid model has been tested successfully in the context of the Multiresolution Adaptive Numerical Environment for Scientific Simulation (MADNESS) application.

Data staging is not always a single copy operation; data may need certain accelerator-specific processing such as transfering them to the device, reformatting them while on the device, and placing them in shared memory[2]. HMPP's CUDA-specific direct share memory operations achieve this. The staging process is also affected by the affinity of data. Certain enhancements to the data residency qualifiers have helped with

[2]The NVIDIA GPU shared memory is not accessible from the host.

data structure that are only "live" on the GPU. Host-device data transfers can be expenssive and advantage needs to be taken of the non-blocking data-transfer opportunities next to the transfers' planning and strategic placement. Improvements against the HMPP asynchronous IO mechanism combined with the mechanism's type-awareness has simplified these tasks.

3 Analysis and Transformation

Static program analysis is a way of automatically analyzing code for the purpose of determining what code restructuring could be performed to gain opmtimal performance. When computationally intensive portions of code have been identified, it is often beneficial to extract the code from the application for implementation on a GPU to improve performance. In order to facilitate developers in restructuring their code and porting computationally intensive aloorithms to a GPU architecture, Titan's PE will incorporate HMPP Feedback and HMPP Wizard into its toolset. The primary purpose of HMPP Feedback is to provide static analysis in text form on code that can be ported or has been ported to a GPU while HMPP Wizard presents the analysis in a graphical user interface (GUI) along with additional diagnoses and analysis to help users perform code transformations.

The HMPP Wizard and Feedback are static analysis tools aimed at detecting source code that prevents GPU parallelization (referred to as diagnosis) and suggests code transformations to increase GPU performance (referred to as advice). In HMPP Feedback, these "diagnoses" and "advice" are provided to developers as a text formatted report to help optimize HMPP generated codelets. The HMPP Wizard serves as a graphical interface to map this analyses to the source code. In addition, for cases where a porting and optimization strategy based on the type of computation is known (i.e. matrix multiplication, convolutions, etc), the Wizard proposes specific advice based on previous knowledge from experts in the computational domain. The Wizard tool uses the same front-end technology as the HMPP compiler, and provides generic programming advice to make the kernels suitable for GPU execution, minimizing performance degradation penalties. The Wizard and Feedback tools serve as a link between the application and the diagnosis, and provides an interactive environment to apply HMPP optimizations.

With both HMPP Wizard and Feedback, every function in the source code file is evaluated as a potential HMPP codelet. A codelet is a pure function that can be remotely executed on a massively parallel accelerator. Loops within a function are considered to be possible GPU kernels. Kernels represent a loop or group of nested loops that define an iteration space and grid of GPU threads. Each loop in a loop nest defines one dimension of the iteration space. The grid of threads defined by the kernel is a subset of the iteration space, and can be used to compute the kernels memory access pattern. Once all of the loops within the potential codelet are analyzed and a GPU grid of threads is computed, a set of analyses are applied to each kernel and performance improvements advice are emitted along with static analyses statistics (i.e. the number of the memory accesses and floating point computation). During the analysis, code between kernels is also taken into consideration when formulating advice. This analysis is only intended to validate computationally intensive code segments as HMPP codelets and optimize HMPP GPU kernels. It does not address CPU-GPU data movement and device allocation optimizations.

In HMPP Wizard, if a computation matches a well-know pattern, the Wizard provides additional advice specific for the type of computation matched. While the advice is meant to be general enough for GPU programming, some of it is targeted at CUDA optimizations such as improving memory coalescing based on memory accesses patterns.

Kernel Validation. When the user wants to port a subroutine to an accelerator using HMPP codelets, ceirtain checks need to be done to the subroutine to ensure that it conforms to the HMPP programming model. The requirements of the HMPP programming model are: (1) that the function codelet has parameters that are either scalars or arrays, (2) the number of parameters are constant and known at compile time, (3) the function does not contain static or volatile variable declarations, (4) the function does not contain references or definitions of function pointers, (5) the function does not use pointer arithmetic, and (6) the function is not recursive. The function may contain references to global data and callsite to other functions within the same source file. In the latter case, the Wizard will check if those functions can be converted to kernels.

Kernel Analysis. After performing the conformance checks, additional analyses is needed to detect the type of parallelism found in a kernel, and see how it can be mapped to a GPU. For this step, a grid of GPU threads analysis is performed. The analysis consists of computing the parallelism status of each loop which can be: (1) sequential: if no parallelism is found, (2) determined parallel: if parallelism is detected by the tool, and (3) specified parallel: if the user inserted a directive specifying that the codelet is parallel. Additionally this analysis will display the shape of the kernel grid for 1D/2D grid kernels, where the X dimension may be the inner loop and the Y dimension may be the outer loop dimension of a loopnest.

Diagnosis and Advice. The HMPP wizard can provide diagnosis to detect code regions that are generating inefficient GPU code. For each of these, diagnosis and advice are presented to the user so he/she can take action to solve the performance issue. An example diagnosis may be the detection of a loopnest that is in a form that can not be normalized which would prevent the HMPP compiler from performing dependence analyses. Another diagnosis may be the detection of branches inside a codelet which would be accompanied by the advice to use masks or split loopnests to eliminate the conditional statements.

Insertion of Directives. Once we have found a subroutine suitable for acceleration, the HMPP Wizard allows the user to select regions of the subroutine to apply HMPP directives. If a directive already precedes the piece of code, it can be selected to insert the directive just after. The user can rerun the codelet analysis by clicking a refresh button which will take into account the newly inserted code. The user may also select a directive and remove it just by clicking on the code window.

Graphical User Interface. The GUI of the Wizard contains a menu, a toolbar, and windows to display the input source code, analyses and advice information. The goal of these windows is to relate and display the source code with the diagnoses, and advice in a user-friendly environment. The File menu performs file related operations such as opening, saving, and closing files. The directive menu can be used to insert or remove directives from the source code. Most of these menu operations can also be done via the toolbar. If a source code is modified, a refresh button can be used to rerun the analyses and advices.

Pattern Matching Analysis. The HMPP Wizard also has the capability of doing pattern matching to compare the memory access footprint of a HMPP codelet to a predefined memory access pattern known by the HMPP wizard which can be used to generate a GPU optimized version of the code. The memory access pattern is defined by the sequences of memory accesses with specific strides combined with a pre-defined pattern for the computations. The array access stride is calculated from the linear expression in the array subscripts that uses the loop induction variable. Currently the HMPP Wizard only supports the array access pattern for the convolution and matrix multiplication.

Figure 2 shows the HMPP Wizard windows when applied to the codelet calcMixH of the S3D application. Initially the kernel analyses were not able to determine that the loop was parallel because of an output dependence. After the user applied a directive specifying that a loop is parallel if a variable is privatized for each thread, the Wizard shows the GPU grid analyses of the new code, and informs the user that there is a conditional statement within the accelerator loop and strategies to hoist it out of the loopnest. Additionally the user can insert more directives using the GUI to keep improving the performance of the code.

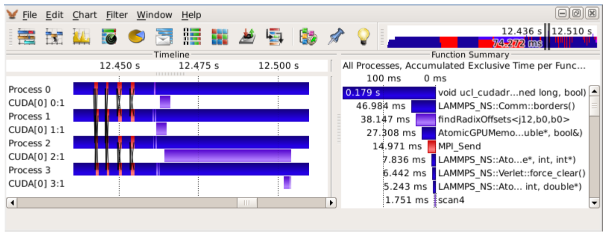
4 Profiling and Tracing

Performance analysis tools help the application developers to fully utilize the resources of growing HPC systems. This is especially true for heterogeneous leadership-class systems that reach new levels of scalability. On the one side, communication patterns, which work well with a few thousands of cores, can become a bottleneck when running hundred thousand cores and more. One aspect of performance analysis tools is to help the application developer to understand the communication patterns in the application and its performance impact. Heterogeneous systems add another aspect to performance analysis.

To fully leverage those systems, the application developer needs to understand the usage of the different resources and the implications of porting complex applications beyond the look at small kernel programs. While performance analysis tools aid the application developers when targeting new large scale heterogeneous systems, those systems also present challenges to the performance analysis tools themselves. What follows is a description of how VampirTrace has been extended for our PE's challenges plus the HMPP Performance Analyzer, which is a product of our partnership with CAPS.

4.1 VampirTrace

The Vampir tool-set is used as performance analysis tools in OLCF-3. We are working together with Vampir's vendor to make this tool-set ready for the targeted OLCF-3 system. Vampir uses program tracing to record a detailed list of events during the execution of an application. Using a set of compiler wrappers for C, C++ and FORTRAN, the application can be built with specific instrumentations. VampirTrace provides instrumentation of the parallel paradigms MPI, and OpenMP/Threads as well as generic recording of function invocations through compiler- or manual instrumentation. Vampir then provides a post-mortem visualization of the program execution based on the recorded trace. This visualization features a set of different displays to help understand the behavior of the application. The analysis for visualization is provided by a parallel server and a GUI application, allowing the processing of large traces. The entire tool-chain is tailored for a scalable parallel analysis. To match the scale of the target OLCF-3 system, additional improvements have and are being incorporated to Vampir. Specific optimizations in the communication behavior of VampirServer now enable the use of more than 10,000 analysis processes. Multiple improvements target the handling of an increasing amount of trace data from hundreds of thousands of processes. Pattern matching based compression will improve the recording, while filtering and the highlighting of irregularities supports the evaluation of large scale traces.



0.179 s	void ucl_cudadr...ned long, bool)
46.984 ms	LAMMPS_NS::Comm::borders()
38.147 ms	findRadixOffsets<j12,b0,b0>
27.308 ms	AtomicGPUMemo...uble*, bool&)
14.971 ms	MPI_Send
7.836 ms	LAMMPS_NS::Ato...e*, int, int*)
6.442 ms	LAMMPS_NS::Verlet::force_clear()
5.243 ms	LAMMPS_NS::Ato... int, double*)
1.751 ms	scan4

Figure 3: Vampir when applied to LAMMPS accelerated with GPU

The other important contribution is the integrated CUDA support in VampirTrace. CUDA-API calls are captured and recorded. GPU events such as kernel execution and memory copies are mapped to CUDA streams. Those events can be invoked asynchronously and are correctly embedded into the timeline of traditional program events. The support for GPU performance counters adds information to the trace. This integrated approach allows analyzing hybrid MPI/OpenMP/CUDA applications as a whole and provides a better picture of the application's performance characteristics than just looking at isolated CUDA kernels. Figure 3 displays a timeline of four MPI processes each with an associated CUDA stream that runs the GPU accelerated version of LAMMPS. With these improvements Vampir provides a comprehensive performance analysis tool for the upcoming OLCF-3 system. It helps the application developers to port and adapt their code to this system and therefore increases its utilization and facilitates the solution of new scientific problems.

It is possible to analyze GPU applications that have been developed with HMPP in Vampir. The code generated by HMPP uses the CUDA runtime library as a backend. The calls to the CUDA library are wrapped by VampirTrace in the same way this is done for manually developed CUDA applications. The same functionality is therefore available for HMPP applications, including memory copies, kernel (codelets) executions, and performance counters. Vampir exposes details on how HMPP maps the codelets to the GPU, but might lose some information about the high level HMPP code. This preservation of high level HMPP semantic is subject to ongoing development. HMPP and VampirTrace both use compiler wrappers for their functionality. Those compiler wrappers have to be chained for the integration. This is done by using *vtcc* as a compiler for *hmpp*.

4.2 The HMPP Performance Analyzer

While certain effort is required to decide on what to offload to the accelerator and the related data-transfer issues, this task is most likely to be succeeded with fine tuning of the kernels runnning on the accelerator, which is often both API (e.g. CUDA) and architecture (e.g. GPU generation) dependent. Our partnership with CAPS has led to the HMPP Performance Analyzer. It assists users with optimizing their HMPP codelets, as opposed to the HMPP Wizard that examines the entire application looking for candidates for conversion to HMPP, conformance, etc.

Users utilize the Performance Analyzer similarly to the Wizard; in fact, the Performance Analyzer specializes the Wizard's infrastructure for its purposes. The users select HMPP codelets from the GUI and then the Performance Analyzer evaluates the (previously gathered) performance figures and provide the users with performance metrics and optimization hints. The Performance Analyzer offers its own, synthetic, metrics such as memory throughput, load/store density, the branch ratios, etc., which are derived from raw metrics that the NVIDIA profiler generates. Apart from the metrics, the Performance Analyzer provides optimization hints, such as loop transformations for minimizing divergence, and generally guides the user on the application of HMPP low-level code-generating directives (`hmppcg`-level) in a spirit similar to that of Wizard with the high-level directives.

5 Debugging

A scalable, hybrid platform aware debugger is an essential component for the PE of Titan that works well on a massive hybrid GPU-based cluster system. We have worked with Allinea to make their debugger scale up to 200,000 cores. Our collaboration with Allinea DDT allows to addresses these requirements by utilizing sophisticated tree topology and tight integration with advanced Crays PE features such as: scalable breakpoints, stepping and program stack queries, scalable process management, scalable visualization of variable values using statistical analysis and prefetching techniques, distributed core file generation with abnormal process termination, and full integration with Crays process launcher. All the above DDT capabilities provide basic building blocks for creating efficient debugger for the PE. Figure 4 shows the time that it takes to set a breakpoint or step over program statements during a debugging session up to 200,000 MPI process. The figure clearly shows that the debugger is scalable.

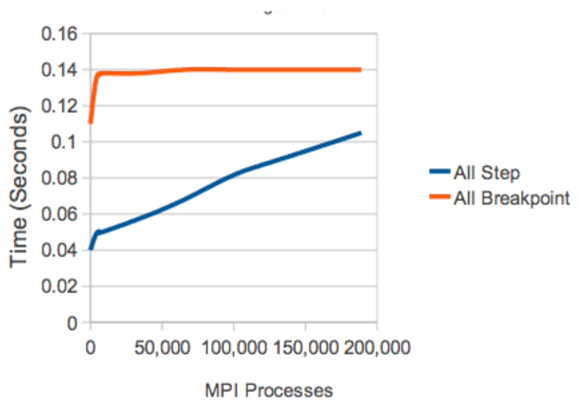


Figure 4: DDT scalable breakpoints and stepping for large MPI process counts in Jaguar XT5

In addition, DDT has enhanced their existing debugger capabilities to support CUDA and the HMPP compiler. The current implementation supports stepping over CUDA kernels, and automatic detection of HMPP fragments, step over HMPP codelets, and report error codes from the HMPP runtime.

Figure 5 shows how we can set up a breakpoint before we enter an HMPP region directive in one of the CAM/SE kernels. The DDT debugger is able to recognize the HMPP directives and step over them correctly.

6 Research

We are currently researching ways to manage the complexity of porting codes to the new Ttitan system. We are currently working on two tools which will help to automate some aspects of the porting work flow such as automating the process of applying transformations and tools for detecting different regions of code in an application that may benefit from the same optimization and porting strategies. For the first task we have developed a tool called HERCULES, and for the latter a similarity analysis tool called Klonos.

6.1 HERCULES

HERCULES is a framework that empowers the application developers with facilities for rapid prototyping of transformations without requiring the developers to be compiler engineers. HERCULES uses two concepts: a pattern and a transformation script. The purpose of the pattern is to identify aspects of an application that exhibit a particular behavior or properties. The pattern and a transformation script comprises a "transformation recipe;" recipes are recorded by HERCULES and can be used for the construction of more complex recipes. HERCULES takes the recipes and applies them against application sources.

HERCULES' integration with the PE presents new opportunities for significant improvements with regards to the pattern's richness and further transformations support. HERCULES' pattern-matching language will be extended in order to take advantage of code transformation tools like ROSE [12] and LLVM [11]. HERCULES' transformations engine will be extended to support the majority of the transformations that will feature GPU-oriented and data motion transformations, and will be integrated in the workflow with the HMPP Wizard and Cray's Apprentice.

6.2 Similarity Analysis

Code replication is a phenomenon that occurs in practice in any large code bases. Developers copy and paste code regions, unknowingly reimplement already existing functionality or apply tools that translate domain languages to production code, each leading to multiple, similar portions of code. Code regions with similar structure (or with a degree of similarity based on a specific metric) are called "code clones" and represent [13, 3] on average 5-20% of production software. Clones pose code maintenance problems and require replication of porting efforts. Four categories of code clones are identified in [13]: (1) identical code fragments (Type I); (2) syntactically identical code fragments with variation in identifiers, literals, types (Type II); (3) copied fragments that have been further modified or changed (Type III); and (4) semantically similar code fragments (Type IV) that are implemented differently.

As porting and tuning codes will benefit from clone detection, we will provide similarity analyses within the PE to detect clones of types I-III. We will combine metrics from static analyses, modeling, and performance measurements to define a similarity metric, and provide different views on how to detect clones in an application. We will use results of related studies [16, 15, 16, 10, 6, 7, 9, 8] to define the metric system to classify families of code clones that are good candidates for porting or optimizations. We have successfully applied this tool to detect code clones in CAM/SE and classify them based on their syntactic similarities.

7 Conclusions

The purpose of this paper has been to present the PE that we are establishing for the upcoming Cray-based Titan system. We continue to collaborate with a number of vendors, namely Allinea, CAPS and TU Dresden, 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. The PE will be at production level upon Titan's arrival.

References

[1] CULA. `http://www.culatools.com/`, 2011.

[2] The HMPP workbench. `http://www.caps-entreprise.com/hmpp.html`, 2011.

[3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, pages 86–, Washington, DC, USA, 1995. IEEE Computer Society.

[4] J. Beyer. OpenMP for Accelerators 1.1. February 2011.

[5] B. Bland. HPC @ ORNL where do we go from here? In *SC10*, 2010.

[6] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. Language-independent clone detection applied to plagiarism detection. In *International Working Conference on Source Code Analysis and Manipulation*, September 2010.

[7] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *International Conference on Software Maintenance*, August/September 1999.

[8] M. Funaro, D. Braga, A. Campi, and C. Ghezzi. A hybrid approach (syntactic and textual) to clone detection. In *International Workshop on Software Clones*, May 2010.

[9] J.H. Johnson. Substring matching for clone detection and change tracking. In *International Conference on Software Maintenance*, September 1994.

[10] R. Koschke. Frontiers of software clone management. In *Froniters of Software Maintenance*, September/October 2008.

[11] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, USA, March 2004.

[12] D. Quinlan. Rose: Compiler support for object-oriented frameworks. In *Proceedings of Conference on Parallel Compilers (CPC2000)*, 2000.

[13] Chanchal K. Roy and James R. Cordy. An empirical study of function clones in open source software. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 81–90, Washington, DC, USA, 2008. IEEE Computer Society.

[14] 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.

[15] R. Smith and S. Horwitz. Detecting and measuring similarity in code clones. In *International Workshop on Software Clones*, March 2009.

[16] A. Walenstein, M. El-Ramly, J.R. Cordy, W.S. Evans, K. Mahdavi, M. Pizka, G. Ramalingam, J. Wolff von Gudenberg, and T. Kamiya. Similarity in programs. In R. Koschke, E. Merlo, and A. Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, April 2007.

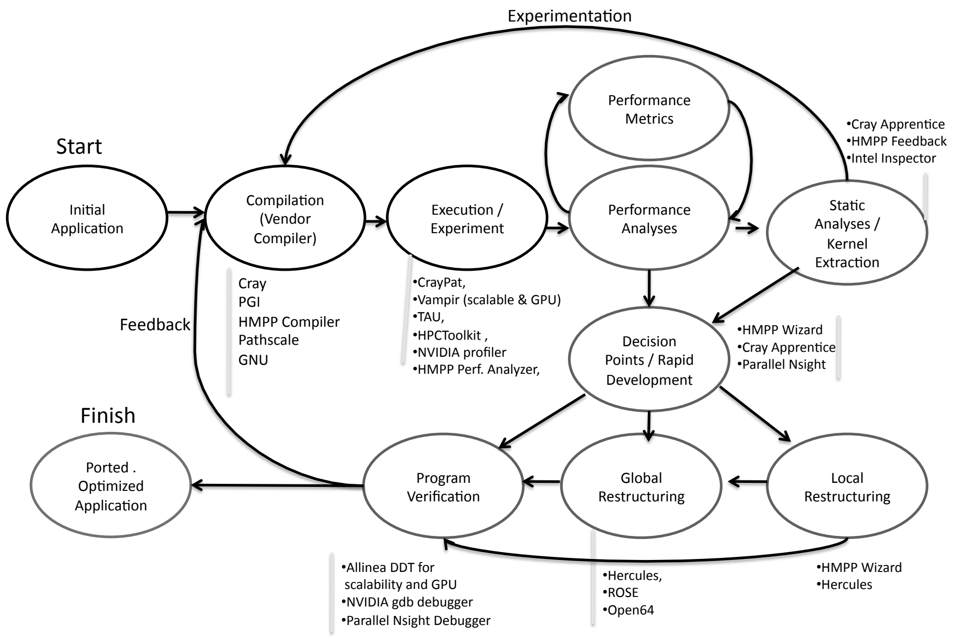
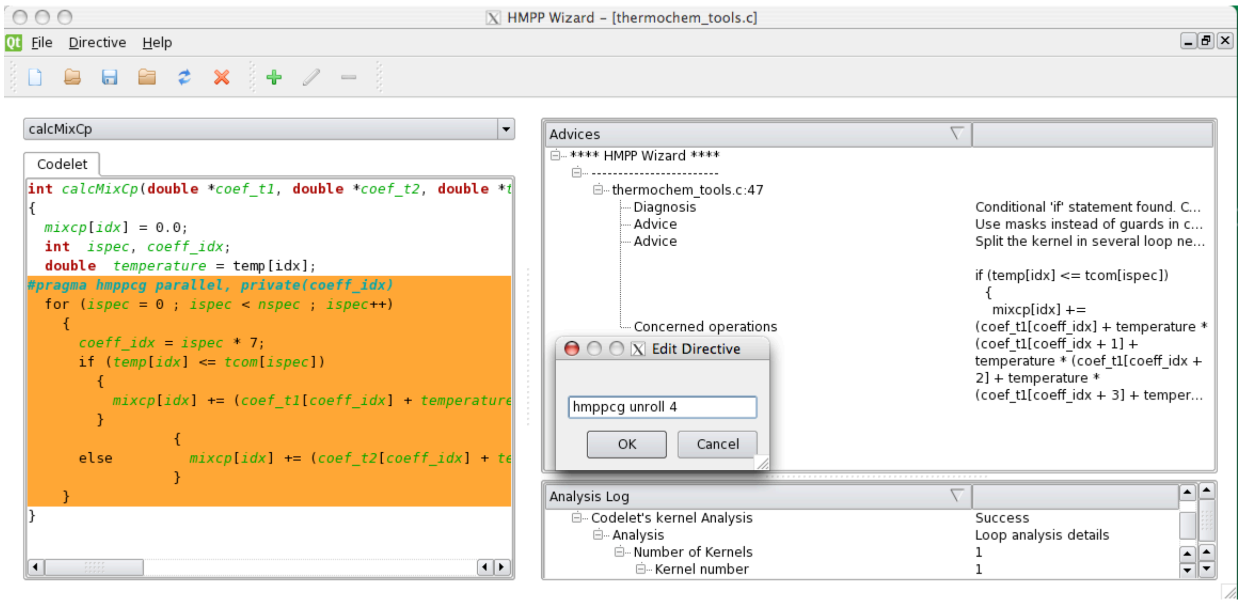


Figure 1: System Diagram



```
int calcMixCp(double *coef_t1, double *coef_t2, double *t
{
  mixcp[idx] = 0.0;
  int  ispec, coeff_idx;
  double  temperature = temp[idx];
#pragma hmppcg parallel, private(coeff_idx)
  for (ispec = 0 ; ispec < nspec ; ispec++)
    {
      coeff_idx = ispec * 7;
      if (temp[idx] <= tcom[ispec])
        {
          mixcp[idx] += (coef_t1[coeff_idx] + temperature
        }
                  {
      else          mixcp[idx] += (coef_t2[coeff_idx] + te
                  }
    }
}
```

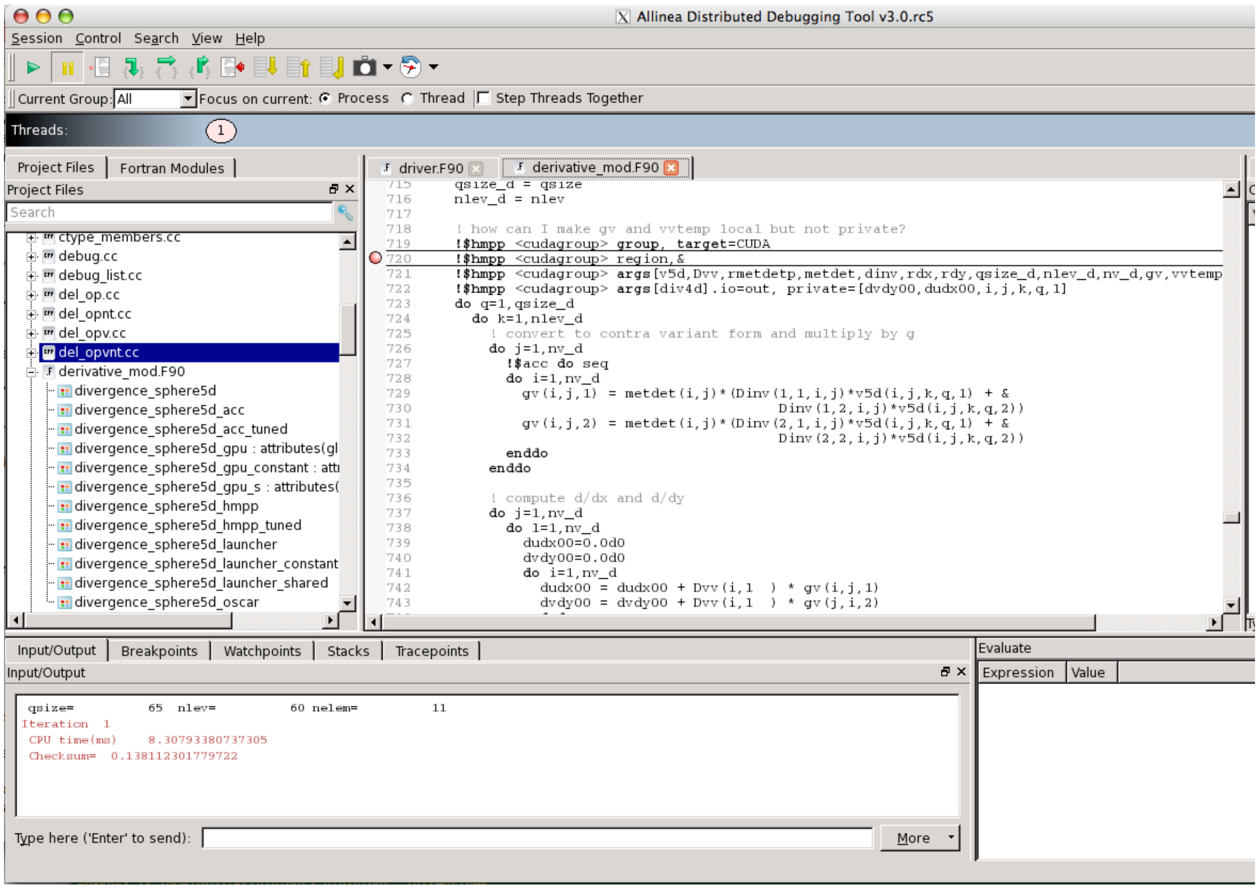
```
Advices
**** HMPP Wizard ****
  -----------------------
    thermochem_tools.c:47
      Diagnosis                      Conditional 'if' statement found. C...
      Advice                         Use masks instead of guards in c...
      Advice                         Split the kernel in several loop ne...

                                     if (temp[idx] <= tcom[ispec])
                                       {
                                         mixcp[idx] +=
      Concerned operations           (coef_t1[coeff_idx] + temperature *
                                     (coef_t1[coeff_idx + 1] +
                                     temperature * (coef_t1[coeff_idx +
                                     2] + temperature *
                                     (coef_t1[coeff_idx + 3] + temper...
```

hmppcg unroll 4

```
Analysis Log
  Codelet's kernel Analysis          Success
    Analysis                         Loop analysis details
      Number of Kernels              1
        Kernel number                1
```

Figure 2: The HMPP Wizard when applied to the codelet calcMixH of the S3D application


Allinea Distributed Debugging Tool v3.0.rc5
Session Control Search View Help
Current Group: All
Focus on current: Process Thread Step Threads Together
Threads: 1
Project Files Fortran Modules
Search
ctype_members.cc
debug.cc
debug_list.cc
del_op.cc
del_opnt.cc
del_opv.cc
del_opvnt.cc
derivative_mod.F90
divergence_sphere5d
divergence_sphere5d_acc
divergence_sphere5d_acc_tuned
divergence_sphere5d_gpu : attributes(gl
divergence_sphere5d_gpu_constant : attı
divergence_sphere5d_gpu_s : attributes(
divergence_sphere5d_hmpp
divergence_sphere5d_hmpp_tuned
divergence_sphere5d_launcher
divergence_sphere5d_launcher_constant
divergence_sphere5d_launcher_shared
divergence_sphere5d_oscar
driver.F90
derivative_mod.F90
Input/Output Breakpoints Watchpoints Stacks Tracepoints
Type here ('Enter' to send):
More
Evaluate
Expression Value

```
     qsize_d = qsize
     nlev_d = nlev

     ! how can I make gv and vvtemp local but not private?
     !$hmpp <cudagroup> group, target=CUDA
     !$hmpp <cudagroup> region,&
     !$hmpp <cudagroup> args[v5d,Dvv,rmetdetp,metdet,dinv,rdx,rdy,qsize_d,nlev_d,nv_d,gv,vvtemp
     !$hmpp <cudagroup> args[div4d].io=out, private=[dvdy00,dudx00,i,j,k,q,l]
     do q=1,qsize_d
       do k=1,nlev_d
         ! convert to contra variant form and multiply by g
         do j=1,nv_d
           !$acc do seq
           do i=1,nv_d
             gv(i,j,1) = metdet(i,j)*(Dinv(1,1,i,j)*v5d(i,j,k,q,1) + &
                                      Dinv(1,2,i,j)*v5d(i,j,k,q,2))
             gv(i,j,2) = metdet(i,j)*(Dinv(2,1,i,j)*v5d(i,j,k,q,1) + &
                                      Dinv(2,2,i,j)*v5d(i,j,k,q,2))
           enddo
         enddo

         ! compute d/dx and d/dy
         do j=1,nv_d
           do l=1,nv_d
             dudx00=0.0d0
             dvdy00=0.0d0
             do i=1,nv_d
               dudx00 = dudx00 + Dvv(i,l  ) * gv(i,j,1)
               dvdy00 = dvdy00 + Dvv(i,l  ) * gv(j,i,2)
```

```
 qsize=          65  nlev=          60 nelem=          11
Iteration  1
 CPU time(ms)    8.30793380737305
 Checksum=  0.138112301779722
```

Figure 5: The DDT debugger when applied to the HMPP codelet divergence_sphere from the CAM/SE application