Performance Tool Integration in a GPU Programming Environment: Experiences with TAU and HMPP

Allen D. MALONY[1,3] , Shangkar MAYANGLAMBAM[1] , Laurent MORIN[2] , Matthew J. SOTTILE[1] , Stephane BIHAN[2] , Sameer S. SHENDE[1,3] , and Francois BODIN[2]

[1]*Dept. of Computer & Information Science, University of Oregon, Eugene, OR 97403*
[2]*CAPS Entreprise, 35000 Rennes, France*
[3]*ParaTools, Inc., Eugene, OR 97405*

Abstract. Application development environments offering high-level programming support for accelerators will need to integrate instrumentation and measurement capabilities to enable full, consistent performance views for analysis and tuning. We describe early experiences with the integration of a parallel performance system (TAU) and accelerator performance tool (TAUcuda) with the HMPP Workbench for programming GPU accelerators using CUDA. A description of the design approach is given, and two case studies are reported to demonstrate our development prototype. A new version of the technology is now being created based on the lessons learned from the research work.

1. Introduction

Multi-core systems with GPU acceleration offer a high performance potential to application developers. Unfortunately, achieving performance improvements with accelerators is challenging due to complexity of the multi-core hardware and their low-level device interface. Programming environments targeting GPU accelerators attempt to hide this complexity by allowing the application developer to work with libraries, special language constructs, or directives to a compiler. The benefit for the programmer is a higher-level abstraction for accelerator programming and protection of their software investment, since the environment takes the responsibility for translating the program to work with different acceleration backends. The challenge for accelerator programming environments is to provide high-level support and flexibility without sacrificing delivered performance. Traditionally the use of performance tools for measurement and analysis allows developers to identify performance inefficiencies and inform optimization strategies. For optimization of GPU-accelerated applications, these tools must 1) be able to measure performance of GPU computations, and 2) be integrated with the high-level programming framework to generate important performance events and meta data for representing performance results to the user. Furthermore, when used in large-scale parallel environments, it is important to understand the performance of accelerators in the context

of whole parallel program's execution. This will require the integration of accelerator measurements in scalable parallel performance tools.

This paper discusses our initial efforts to integrate the TAU Performance System® [4] and the HMPP Workbench [2]. We focus on the use of the prototype TAU CUDA measurement interface (TAUcuda [5] within HMPP and the model for inserting TAU instrumentation in the HMPP-translated code to best present a performance picture of the resulting application execution. Two case studies are presented to demonstrate the approach.

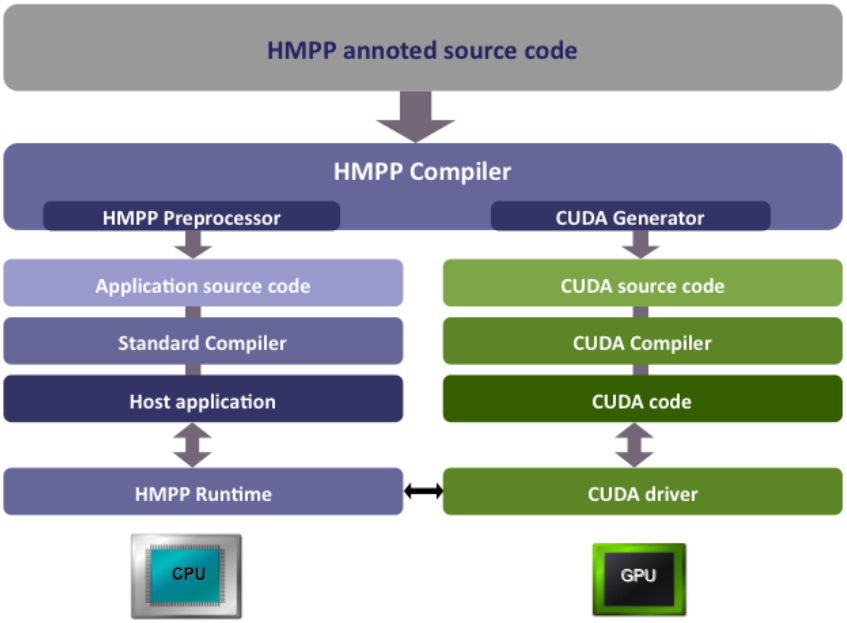

Figure 1. HMPP Workbench compilation for applications targeting CUDA.

2. Design Approach

The objective of a high-level programming environment is to insulate the developer from dealing with low-level concerns. In the case of accelerator programming, the HMPP Workbench offers a directive-based approach for C and Fortran languages to specify *codelets* for execution on accelerator devices and *callsites* in the host program where codelets will be invoked. HMPP operates as a source-to-source translator, adding all the necessary host-side code to interface with the accelerator, and generating target-specific code depending on accelerator type. Figure 1) shows the two compilation paths needed to build HMPP applications with CUDA as the target accelerator. The HMPP execution model allows for asynchronous CPU/GPU execution and managed data transfers, utilizing the functionality provided in the CUDA driver interface.

To evaluate the performance of an HMPP application, it is necessary to make measurements of important execution events and to relate the performance data back to the

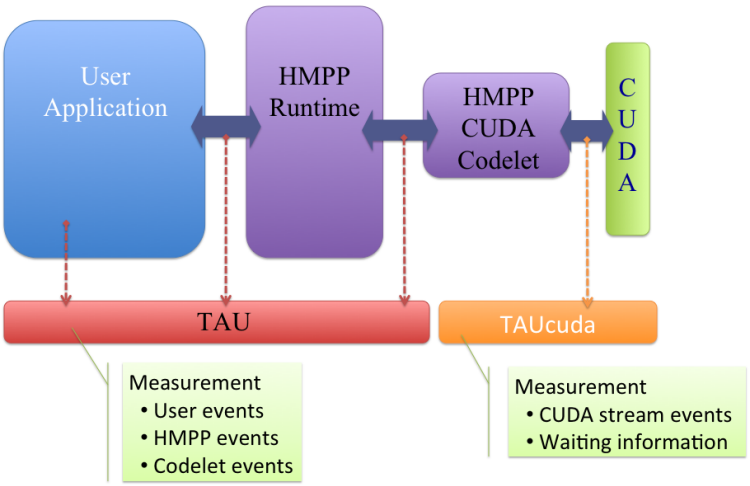

Figure 2. Integration of HMPP, TAU, and TAUcuda instrumentation and measurement support.

HMPP programming abstractions. As shown in Figure 2, there are three levels in the HMPP framework to instrument to capture the full picture of application execution: application code, HMPP runtime, and CUDA codelets. Although HMPP is a source-to-source translator, the application developer is ill-equipped for performance instrumentation on their own, much less knowledgable of the event semantics between levels. The advantage HMPP brings for performance integration is in the automation of instrumentation, designed specifically to capture necessary event information to present a full performance view.

Instrumentation relies on an underlying measurement infrastructure. We chose the TAU Performance System for HMPP performance measurements, given its robust capabilities for profiling and tracing of parallel applications. However, TAU only solves the problem of CPU-side measurements. Some other technology was needed for CUDA measurements. Luckily, our concurrent work on the prototype TAUcuda system offered the missing piece. (See [5] for more information on TAUcuda.)

Figure 2 identifies which performance events are measured by TAU and TAUcuda, respectively. HMPP is responsible for placing all instrumentation appropriately in the generated code and in the HMPP runtime system. The application build chain with instrumentation included is shown in Figure 3. Notice that TAU's instrumentation tool can be used at the start to generate events for (non-HMPP) application-level routines. Measurements of these events are important because they provide an application-level *context* for HMPP-related and CUDA-related events. We use the term *HMPP-TAU* to refer to entire integrated performance tool chain.

Having integrated the different instrumentation and measurement facilities, the goal for HMPP performance analysis was to track the events unfolding during codelet execution and reconstruct a high-level view to highlight performance problems. One of the challenges introduced was HMPP's support for asynchronous codelet execution as well as memory transfers. This exposed the need for new mechanisms to be developed in TAU to correctly maintain performance data from concurrent tasks. A workaround using virtual threads was developed by CAPS Entreprise to deal with problem, but a more robust solution will be necessary for going forward.

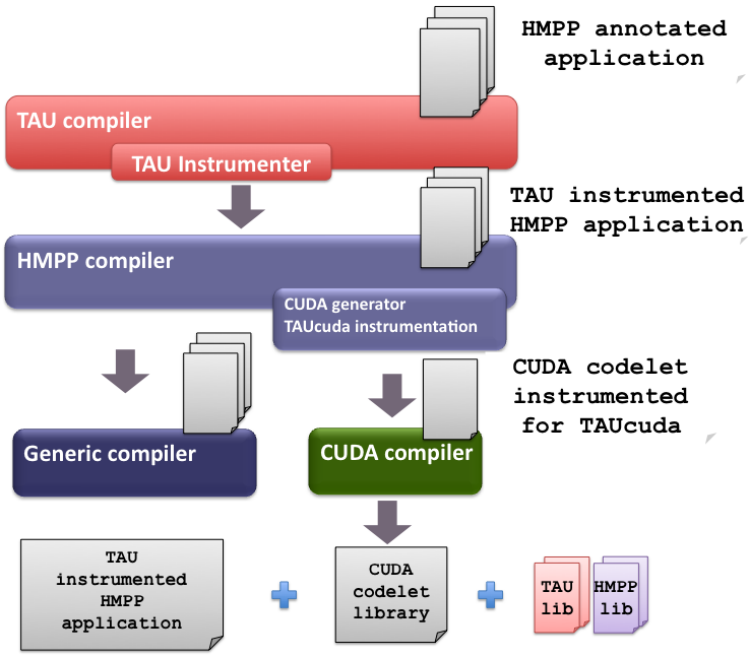

Figure 3. HMPP application build chain with instrumentation enabled.

The issue had to do with HMPP's model of computation and how its runtime system maintained the codelet abstraction during execution. Effectively, it uses a single thread to interface to a GPU device, but manages codelets separately on GPU streams. TAU requires a dinstinguishable thread ID to separate HMPP codelet-specific events. However, since only one real thread is used, a virtual thread ID needed to be provided for TAU events when the HMPP runtime is working on behalf of a specific codelet.

3. Case Studies

Two benchmarks were used as case studies to test the functionality of HMPP-TAU. The first was a implementation of Conway's *Game of Life* [3] (GoL) using HMPP. Figure 4 lists the entire HMPP program, showing the codelet callsite in the main routine on the left panel and the codelet specification with two parallel loop kernels on the right. We used HMPP-TAU to instrument the GoL application and ran several experiments with differenent problem sizes (number of cells) to see the performance effects. The results are shown in Figure 5. Notice there are different TAU events listed representing different (CPU-side) HMPP instrumention points. Those prefixed by `hmpp` correspond to the HMPP runtime layer, while those prefixed by `codelet_` correspond to the codelet interface. Because the HMPP codelet is launched synchronously, the `codelet_wait` event effectively contains the execution time of the two kernels. The `hmppStartCodelet` event is the HMPP runtime library event encapsulating the `codelet_wait`. The reults show how the increasing problem size results in larger kernel execution times, as well as larger times for data transfer.

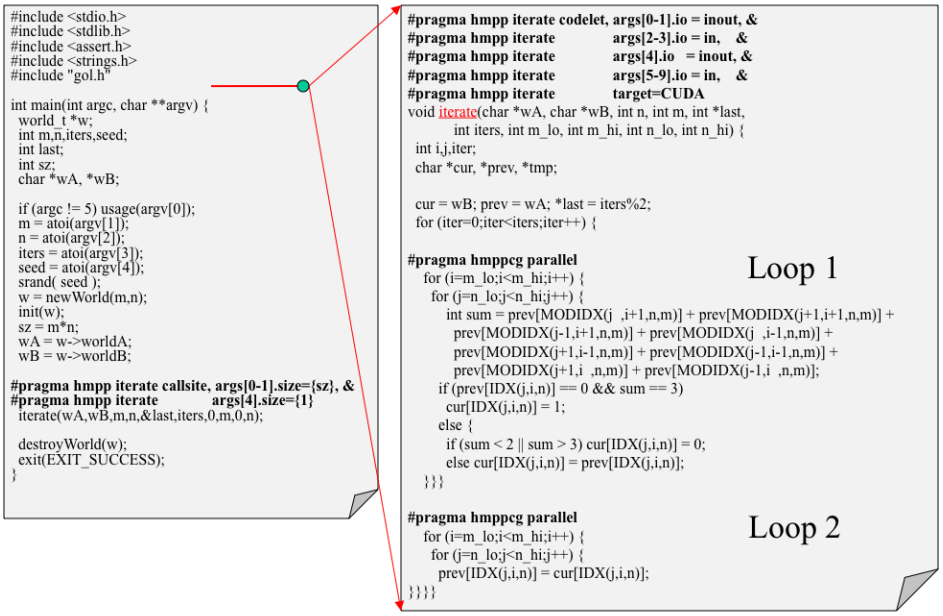

Figure 4. Game of Life HMPP source code.

TAU Event	calls	Computation time(ms) for varying input matrix size				
		1000X1000	2000X2000	3000X3000	4000X4000	5000X5000
hmppStartCodelet	1	7931	25506	95711	152348	345741
hmppGetAvailableHWA	1	2765859	2702177	2705051	2769246	2714853
hmppReleaseDevice	1	50235	2732	2831	44236	2923
hmppWriteDataToHWA	10	1318	1596	2064	2694	3488
hmppAllocateInputOnHWA	7	1234	1235	1234	1241	1277
hmppReadDataFromHWA	3	1718	3261	5648	8894	12811
hmppAllocateInOutOnHWA	3	1218	1357	1256	1302	1277
hmppStartHMPP	1	9	11	11	12	11
codelet_wait	1	5566	22925	93184	149688	343263
codelet_readDataFromHWA	3	590	2021	4320	7522	11534
codelet_writeDataToHWA	10	121	371	808	1430	2235
codelet_allocateInOutOnHWA	3	81	88	99	117	138
codelet_start	1	104	127	134	131	131
codelet_allocateInputOnHWA	7	0	0	1	1	1

Figure 5. Game of Life scaling results.

The second benchmark is a standard vector matrix multiplication used to demonstrate the advantage of overlapping GPU kernel computation with data input/output transfers. Consider the two cases portrayed in Figure 6. The *Sequential* case requires `Vin` vector and the `Min` matrix to be first uploaded to the GPU device (upper part of picture) before the kernel computation can begin (lower part of picture). Only a single HMPP codelet would be used. Writing back of results (`Vout`) can be pipelined with the kernel execution to a limited extent in the codelet. The *Overlapped* case breaks up the `Min` data transfer into columns and overlaps it with vector-column multiplication and `Vout` results transfer. The result is more efficient pipelined execution with greater performance. However, two HMPP codelets are required to make this happen.

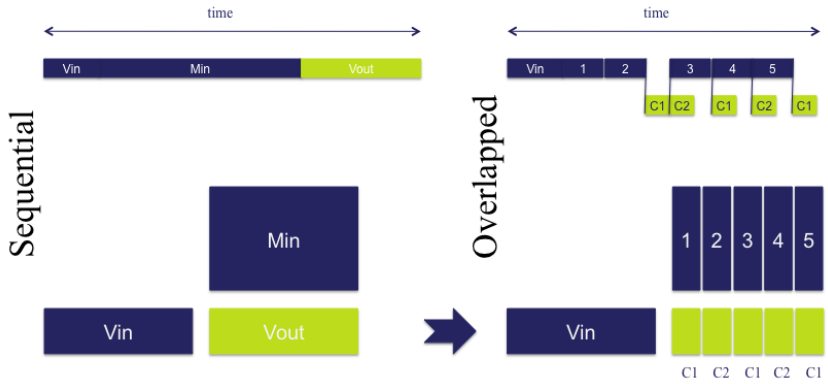

Figure 6. Vector-matrix multiplication benchmark: sequential and overlapped.

Performance analysis for this benchmark focuses on identifying the column blocking for most efficient overlap and minimal total execution time. The performance experiments demonstrate the ability of HMPP-TAU to capture events at different levels for profiling and trace analysis. In Figure 7, we see a display from TAU's ParaProf [1] tools listing the exclusive time for all events, including the TAUcuda events. Profile results allows event time ratios to be compared to determine optimal blocking parameters. ParaProf can conduct an analysis across multiple experiments with different parameters and display the results to the developer.

TAU is also able to capture the HMPP-TAU events in an execution trace. This enables the temporal behavior of the events to be observed in order to highlight event relationships and ordering. However, we needed to do a little hand massaging of the performance data to separate the events into virtual thread traces. Figure 8 uses the Jumpshot [6] tool to display events from the HMPP runtime and codelet levels for the overlapped vector matrix benchmark. One can see the main HMPP thread at the top setting up the computation and kicking off the codelets. Each codelet executes asynchronously of the other, but their execution is coordinated by the main HMPP thread. The trace visualization allows us to see the data transfer of one codelet overlap with the kernel execution of the other. Of course, profile and trace analysis tools can be used together in performance investigation, as shown in Figure 9.

4. Conclusion

We developed the early HMPP-TAU prototype reported on here in just a few weeks of efforts, after the first version of the TAUcuda tool became functional. While the results are quite encouraging, the integration of the three components – HMPP, TAU, and TAUcuda – exposed several issues that need more consideration. It is important that the HMPP programming abstraction and execution model be reflected in the performance views being delivered by HMPP-TAU. The support for asynchronous modes in the HMPP runtime system and codelet execution (not to mention CUDA's over nuances) was a challenge to merge with TAU's way of handling events in a multi-threaded program. We used retrofits to get things working successfully, but a better designed and more robust solution is re-

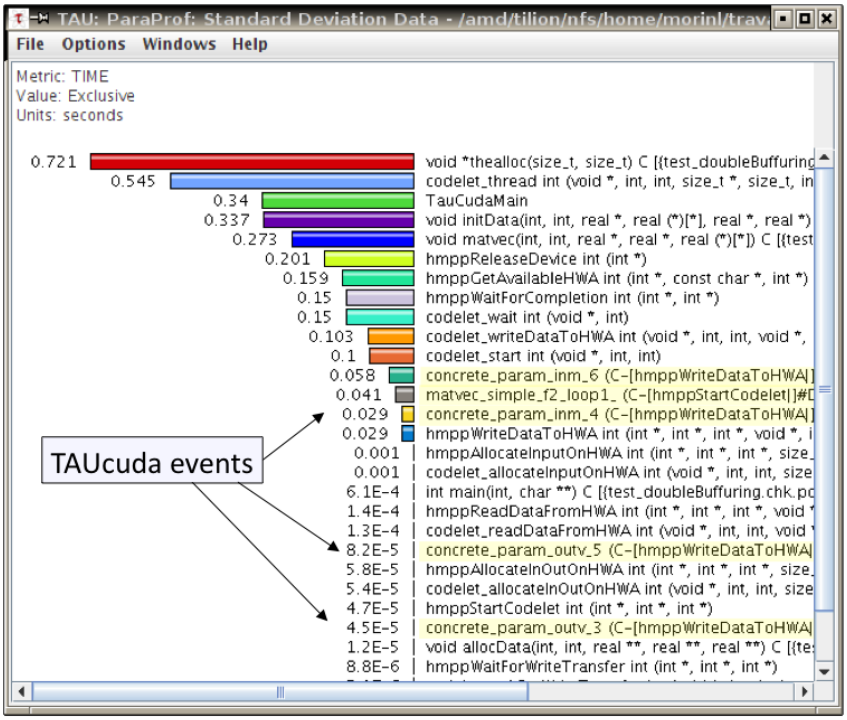

Figure 7. Profile of multiplication benchmark showing events from all instrumentation levels.

quired. The good news is that the experience gained in identifying HMPP events, automating instrumentation, and using TAU's measurement API will all translate forward into future developments.

Our plan is to re-engineer HMPP-TAU in the coming months. The TAUcuda tool is being re-developed presently to use new technology from NVIDIA for access to CUDA library, driver, and kernels execution events. HMPP-TAU will benefit directly from this work. Although the benchmarks shown here are basic, HMPP can be used for parallel applications targeted to large GPU clusters. We intend to HMPP-TAU to be used in such scenarios.

Acknowledgments

This research was supported by the U.S. Department of Energy, Office of Science, under contract ER25933 and the NVIDIA Professor Partnership grant at the University of Oregon.

References

[1] R. Bell, **A. Malony**, and S. Shende. A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *European Conference on Parallel Processing (EuroPar)*, volume LNCS 2790, pages 17–26, September 2003.

Figure 8. Trace of multiplication benchmark showing temporal events relationships.

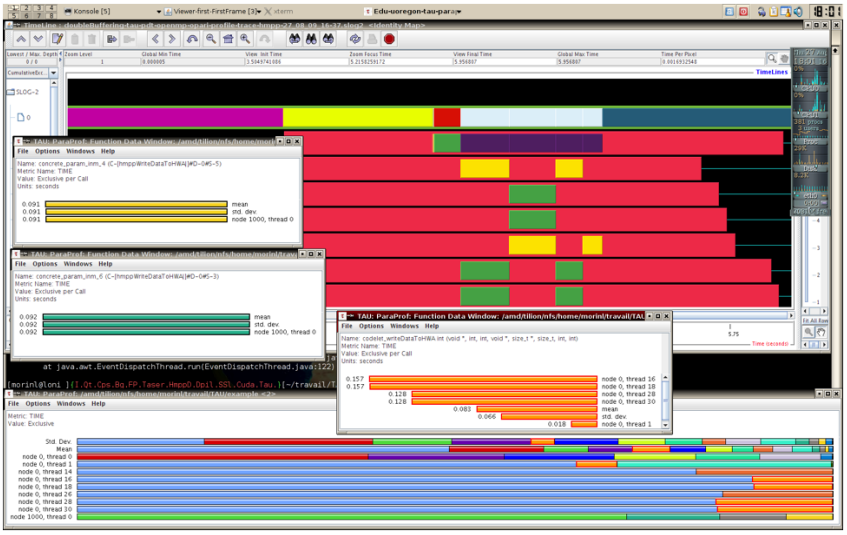

Figure 9. Combined HMPP-TAU performance analysis environment.

[2] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[3] M. Gardner. Mathematical Games: the Fantastic Combinations of John Conway's New Solitaire Game "Life". *Scientific American*, 223:120–123, October 1970.

[4] A. Malony, S. Shende, A. Morris, S. Biersdorff, W. Spear, K. Huck, and Aroon Nataraj. Evolution of a Parallel Performance System. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *2nd International Workshop on Tools for High Performance Computing*, pages 169–190. Springer-Verlag, July 2008.

[5] S. Mayanglambam, A. Malony, and M. Sottile. Performance Measurement of Applications with GPU cceleration using CUDA. In *Parallel Computing (ParCo)*, September 2009. To appear.

[6] O. Zaki et. al. Toward scalable performance visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(3):277–288, Fall 1999.