Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 00 (2013) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2013

High-level support for hybrid parallel execution of C++ applications targeting Intel® Xeon Phi™ coprocessors

Jiri Dokulil[a,*], Enes Bajrovic[a], Siegfried Benkner[a], Sabri Pllana[a], Martin Sandrieser[a], Beverly Bachmayer[b]

[a]*Research Group Scientific Computing, University of Vienna, Austria*
[b]*Software and Solutions Group, Intel GmbH, Germany*

Abstract

The introduction of Intel® Xeon Phi™ coprocessors opened up new possibilities in development of highly parallel applications. Even though the architecture allows developers to use familiar programming paradigms and techniques, high-level development of programs that utilize all available processors (host+coprocessors) in a system at the same time is a challenging task.

In this paper we present a new high-level parallel library construct which makes it easy to apply a function to every member of an array in parallel. In addition, it supports the dynamic distribution of work between the host CPUs and one or more coprocessors. We describe associated runtime support and use a physical simulation example to demonstrate that our library can facilitate the creation of C++ applications that benefit significantly from hybrid execution. Experimental results show that a single optimized source code is sufficient to simultaneously exploit all of the host's CPU cores and coprocessors efficiently.

Keywords: Intel Xeon Phi coprocessor; C++; hybrid execution; Intel Threading Building Blocks; Offload

1. Introduction

The Intel Xeon Phi coprocessor is a novel contender in the heterogeneous computing market. Due to its conceptual similarity to traditional shared-memory systems and a familiar instruction set, it promises to be an interesting alternative to other hardware (e.g., GP-GPUs) for accelerating applications. However, even though established parallel programming frameworks, tools and libraries (e.g., OpenMP, TBB) can in many cases be used seamlessly on this architecture, the distribution of work within a hybrid (host+coprocessor) system is still a very challenging task requiring manual tuning.

In this paper, we present the design and prototype implementation of a hybrid library that can distribute work dynamically to all available processing units (host-cores and coprocessor-cores) in a system. We utilize standard C++ meta-programming techniques to provide a familiar and easy-to use construct (parallel-for) that enables hybrid execution on multiple heterogeneous processing resources.

There are several libraries and tools that support the Xeon Phi coprocessor. However, only few directly support hybrid execution [1, 2, 3] – a scenario where one large task is split into smaller pieces which are executed on the

*Corresponding author.
E-mail address: jiri.dokulil@univie.ac.at.

host and (possibly multiple) coprocessors at the same time. It is possible to statically (i.e., before the computation starts) divide the work into several groups and then run one group on the host and each of the remaining groups on a coprocessor. However, this solution is not suitable in situations where the time needed to complete each work-unit cannot be determined in advance. For this reason, we implemented a dynamic work allocation scheme.

The main goal of our work was to provide high-level support for hybrid execution of parallel loops, without forcing developers to abandon modern C++ techniques. We further aim at making applications easier to debug, deploy, and maintain, while still enabling good performance.

Our main contributions presented in this paper are the following:

- We have created a library that supports hybrid execution in C++ applications using Xeon Phi coprocessors. The library interface is easy to use and follows current trends in C++. The algorithms are used the same way as standard C++ and TBB algorithms. Only one implementation of the user's action is necessary – no need to write second source code for the Xeon Phi. We follow the design of TBB and support integration with other TBB code that is being executed on the host and the coprocessor(s).
- We present a new approach of developing code for coprocessors where template metaprogramming is used together with the support provided by the offloading compiler to build specialized algorithms from single C++ source code.
- We show how our library can be utilized for development of a non-trivial fluid simulation code. The example has not been specially selected and tuned to be the best possible fit to our library – it was selected as a scenario where a real developer may be interested in using our library for non-trivial work.
- We demonstrate the performance of the library for several different system configurations featuring multiple CPUs and Xeon Phi coprocessors. Our hybrid implementation is compared to a parallel implementation with the same level of optimization.

2. Hybrid library

The extensive C++ support and the design principles of TBB gave us the idea to further extend the standard `for_each` function template, which executes a function on each member in a container. The TBB library provides `parallel_for_each` that performs the same functionality in parallel. We decided to create the `offload_for_each` function template to perform the same work, but in a hybrid way.

Since there is no shared memory between the host and the coprocessors, it is not possible to fully mimic the behavior of `for_each`. What `offload_for_each` does is: a copy of the function object is created on each of the processors (the host and the coprocessors). Then, the host's copy is executed on some of the items in the input sequence in parallel. The rest of the items are sent (serialized, transferred with DMA and deserialized) to the coprocessors where a local copy of the function object is executed on items in parallel. The result of the function object invocation is then sent back to the host and stored in the original sequence. Each of the items is sent to just one coprocessor unless it is processed by the host. In this case it is not transferred at all – the function is executed in-place. Note, that items are not sent one item at a time. Multiple items are sent together as larger blocks and double buffering is used to improve overall performance. The execution is synchronized using messages and data is transferred via DMA. For both we utilize the SCIF [4] library.

3. Smoothed particle hydrodynamics

We have chosen smoothed particle hydrodynamics [5] simulation of a nebula (gas cloud in space) as an example to test our library. This is a very general method for fluid simulation. The idea is to view the liquid as a set of particles that carry the physical properties of the liquid. The position of the particles may arbitrarily change over time. So, it is not a grid based technique. The actual physical properties of the simulated liquid at any point is derived from the physical properties of the particles that lie near that point. The influence of each particle is defined by a kernel function. We use a Monagnan cubic spline [6] as the kernel function. The advantage of this kernel is the fact, that beyond certain radius h, the value of the function is 0. So, to evaluate properties of the liquid at any point, we only need to consider particles that are closer than h. A spatial index is used to speed up lookup

Table 1. SPH performance for 1 000 000 particles using various configurations of the host and coprocessors.

system configuration	**pipelined execution**	**total time**	**work done by coprocessor**	speedup
serial (1 core)	no	17174.40	0%	1
host (all cores)	yes	1834.72	0%	9.3
host (all cores)	no	1876.57	0%	9.2
coprocessor only	no	3282.94	100%	5.2
host + 1 copr.	yes	885.63	54%	19.4
host + 1 copr.	no	927.31	55%	18.5
host + 2 copr.	yes	628.13	71%	27.3
host + 2 copr.	no	678.76	71%	25.3

of the particle's neighbors. An interesting effect of this property is the fact that the time to process each particle varies from particle to particle, since it is proportionate to the number of neighbors that are close.

In addition to the simulation, the result of each step is also rendered into an image. But volume rendering is computationally not trivial. Hence, we also tried pipelining execution of simulation and rendering steps.

4. Experiments

The experiments have been performed on a machine with two six-core Intel Xeon CPUs (X5680, 12M Cache, 3.33 GHz, Hyper-Threading) and two Xeon Phi coprocessors (pre-production model, 61 cores, 240 worker threads). The source codes were compiled with the Intel C++ Composer XE compiler set to O3 optimization level. All reported times are wall clock times, measured by the host machine. The stop watch was always started just before the simulation, after the initial setup. It was stopped right after the last frame got rendered.

4.1. Configuration variants

In this section, we will use the same problem setup but different configurations of the run-time. We used one million particles and the video resolution was set to 100x100 pixels. Table 1 shows performance results for different library and machine configurations. In our experiments we have used all available cores of the host, except for the serial reference and coprocessor-only experiments.

We observe a significant performance improvement when one Xeon Phi card is utilized. The addition of the second card further improves performance, although not by such a large margin. There are two reasons for this: The first is the Amdahl's law. The preparatory and cleanup phases of each simulation step are done sequentially on the host. The second reason is the fact, that both cards perform DMA transfers at about the same time. For example at the beginning of the execution, when the data is transferred to all of the cards.

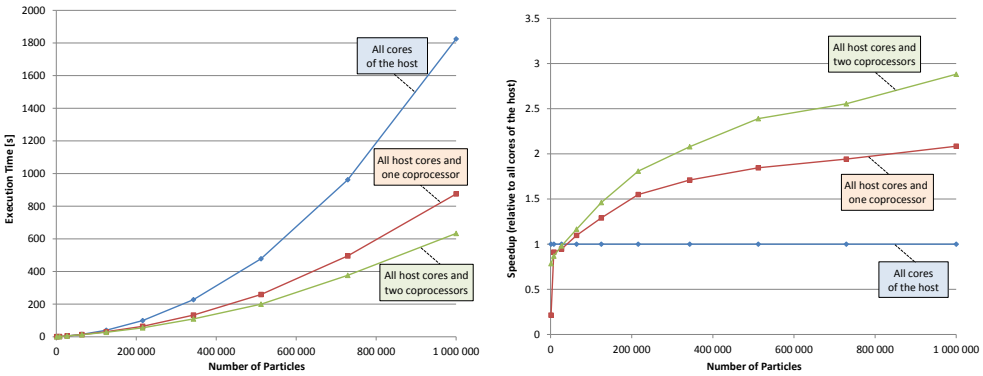
As you can see from the results shown in Table 1, running the code on one coprocessor only (without host cores), does not provide performance improvement over parallel host-only or hybrid execution. This observation can be seen as one of our main motivations for investigating a *hybrid* library. Similarly to the presented example, many existing applications still feature inherent sequential parts better suited for host processors with high single-thread performance. In such cases, the utilization of specialized resources (powerful host-cores in conjunction with highly-parallel coprocessors) can lead to performance improvements.

Another interesting result is the 2 to 8% speedup (depending on the configuration) that was achieved by pipelining the simulation and rendering steps. When the execution is pipelined, the rendering can use cores that would otherwise be idle during the preparatory phase of the simulation.

We also have measured the overhead of the library. It depends heavily on the amount of work that the function object executes (on average) for each data item, as well as on the size of the data. For trivial functions and small data, overhead can be high. In extreme cases, the host could complete the whole work before the offloading support even manages to initialize. For the presented SPH example, overhead was as low as 5%. This includes the time coprocessor cores were idle or executing library code (i.e., working, but not executing the user's function).

4.2. Different problem sizes

Figure 1 depicts SPH performance for various system configurations and numbers of particles. All variants use pipelined rendering. Configurations differ by the number of used coprocessors. Figure 1(a) depicts execution



(a) Execution Time (b) Speedup

Fig. 1. SPH performance for various system configurations and numbers of particles {1k, 8k, 27k, 64k, 125k, 216k, 343k, 512k, 729k, 1000k}.

time in seconds for each experiment and Figure 1(b) depicts the same data as speedup relative to the host-only variant. We observe that for small data sizes, the overhead created by the communication with coprocessors outweighs possible performance gains. This is applicable for scenarios until around 50 thousand particles. For higher numbers of particles, we observe performance improvements when coprocessors are added to the execution. Best performance is achieved when all available host- and coprocessor-cores are utilized.

5. Conclusions

In this paper, we have explored an idea of a new type of offloading library for Intel Xeon Phi coprocessors. It is built around standard C++ paradigms and providing hybrid execution with dynamic work allocation. The library aims at facilitating the utilization of processing resources in modern hybrid systems (host+coprocessors) via high-level programming support. We evaluate the concept by implementing a non-trivial fluid simulation code. Experimental results indicate the viability of our approach. Hybrid execution achieved significant application speedups of up to 27.3x, even with a work-load not ideal for the coprocessor's architecture. We further discovered that our library can facilitate hybrid application development, debugging and optimization due to it's familiar programming methodology. Since our results show that the concept is viable, we have decided to implement a full library based on the same principles, but with a reworked core that provides even better integration with TBB and eliminates the need for management threads on the host.

Extended version: A more detailed description of our work is also available as e-print via arXiv [7].
Acknowledgement: This work was partially supported by the EU FP7 grants AutoTune (no. 288038) and Peppher (no. 248481). We thank the Intel corporation for hardware and software support.

References

[1] J. Dongarra, M. Gates, Y. Jia, K. Kabir, P. Luszczek, S. Tomov, MAGMA MIC, `http://icl.cs.utk.edu/magma/software/index.html`.
[2] Intel, Using Intel Math Kernel Library on Intel Xeon Phi coprocessors, `http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mkl_userguide_lnx/GUID-108D4F3E-D299-4D35-BAED-ECC5B48DA57E.htm`.
[3] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23 (2011) 187–198.
[4] Intel, Intel Xeon Phi Coprocessor System Software Developers Guide, `http://www.intel.de/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-software-developers-guide.pdf` (2012).
[5] R. A. Gingold, J. J. Monaghan, Smoothed particle hydrodynamics - Theory and application to non-spherical stars, Monthly Notices of the Royal Astronomical Society 181 (1977) 375–389.
[6] J. J. Monaghan, J. C. Lattanzio, A refined particle method for astrophysical problems, Astronomy and Astrophysics 149 (1985) 135–143.
[7] J. Dokulil, E. Bajrovic, S. Benkner, S. Pllana, M. Sandrieser, B. Bachmayer, Efficient hybrid execution of C++ applications using Intel Xeon Phi coprocessor, CoRR abs/1211.5530, `http://arxiv.org/abs/1211.5530`.