Deploying OpenMP Task Parallelism on Multicore Embedded Systems with MCA Task APIs

Peng Sun, Sunita Chandrasekaran, Suyang Zhu and Barbara Chapman
Department of Computer Science, University of Houston, Houston, TX, 77004, USA
{psun5,sunita,zsuyang,chapman}@cs.uh.edu

***Abstract*—Heterogeneous multicore embedded systems are rapidly growing with cores of varying types and capacity. Programming these devices and exploiting the hardware has been a real challenge. The programming models and its execution are typically meant for general purpose computation; they are mostly too heavy to be adopted for the resource-constrained embedded systems. Embedded programmers are still expected to use low-level and proprietary APIs, making the software built less and less portable. These challenges motivated us to explore how OpenMP, a high-level directive-based model, could be used for embedded platforms. In this paper, we translate OpenMP to Multicore Association Task Management API (MTAPI), which is a standard API for leveraging task parallelism on embedded platforms. Results demonstrate that the performance of our OpenMP runtime library is comparable to the state-of-the-art task parallel solutions. We believe this approach will provide a portable solution since it abstracts the low-level details of the hardware and no longer depends on vendor-specific API.**

1. Introduction

Software developers for embedded systems tend to deploy task parallelism to get better performance. The automotive application is a good example. The state-of-the-art automobile MCU need to handle different signals and executions from many various aspects, such as electronic controls units for valves, fuel injection, sensors for various aspects of information and driver assistance. Each of these operations could be classified as a task. Programming an embedded system and expressing task parallelism is a real challenge, the use of low-level proprietary APIs makes the development process tedious and error-prone. Besides, hardware designers pack more and more specialized processors to embedded devices and System on Chips (SoCs), such as DSPs, FPGAs, and GPUs. Each of these hardware architectures has the different instruction set itself and sometimes these architectures are co-located with each other, such as the Tegra processor from NVIDIA, which has an ARM and GPUs. This makes the software development even more complicated, and the learning curve further steep.

Other issues include the application developers tackling the distribution of computations and tasks among different computation elements and solving the synchronization and communication problems that will arise. To address some of these challenges, we have been exploring several programming techniques. One of them is OpenMP[1], a pragma-based programming model that allows the software developers to incrementally parallelizes sequential codes. OpenMP's task model provides data parallelism or task parallelism without the need to handle lower-level threading libraries or OS level resources. OpenMP directives will be ignored by a compiler if the compiler does not support OpenMP. In 2013[2], OpenMP released 4.0 specification that allows a program to offload part of the computation from the host (such as CPU or ARM) to the devices (such as GPU, DSP). Early studies on OpenMP 4.include [3], [4].

Though OpenMP could be potentially considered as a candidate for programming embedded systems, there are still significant impediments regarding its usage. Unlike common features in the general-purpose computation devices, embedded systems are typically short of hardware resources and OS-level support. As an instance, most of the OpenMP compilers translate the pragmas into parallel codes with OS-level threading libraries. Cache coherence is also expected by the OpenMP compilers. Not surprisingly, many embedded systems lack these resources and features.

This paper focuses on investigating how such a high-level model, OpenMP could be used for non-conventional systems. To make this feasible, we have considered APIs defined by the Multicore Association (MCA)[5] as the translation layer for the OpenMP runtime library. MCA is a non-profit industry consortium that is formed by a group of leading semiconductor companies and academics, to define open standards for multicore embedded systems. There are three major specifications established by the MCA, the Resource Management API (MRAPI), the Communication API (MCAPI) and the Task Management API (MTAPI). We have explored mapping of OpenMP runtime on MRAPI and the suitability of using MCAPI across embedded nodes in our previous work [6], [7]. Our results demonstrated that such an approach, i.e. translating OpenMP to MCA API, has a lot of potential both in terms of programmability and portability. Hence in this paper we have decided to explore further the translation of OpenMP's task model to MTAPI. The Figure 1 illustrates the solutions stack we proposed for this project.

The main contributions of this paper are: a) Explore the suitability of mapping OpenMP onto embedded systems with MTAPI industry standards. b) Use MTAPI to abstract the lower-level system threading libraries and synchronization primitives, providing OpenMP sufficient parallel execution support at the runtime level. c) Show the effectiveness of our OpenMP runtime library, by measuring OpenMP task overheads and comparing the performance of benchmarks with other leading OpenMP compilers.

Section 2 presents the related work, a brief introduction of OpenMP and MCA is introduced in Section 3, Section 4 discusses the design and implementation of OpenMP's translation to MTAPI. Section 5 and Section 6 discusses results, conclusion and future work.

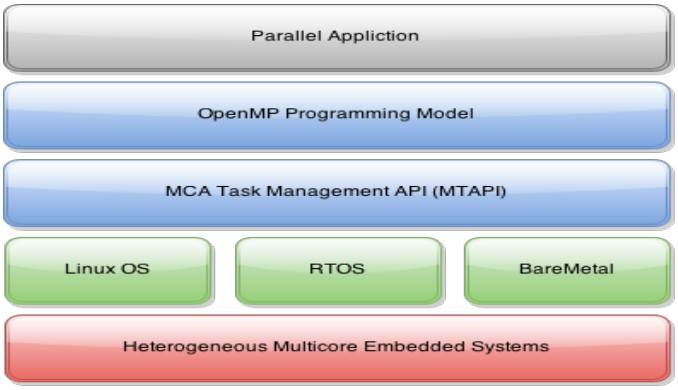

Fig. 1: Project Solution Stack

2. Related Work

This section discusses related efforts on parallel programming models for embedded systems and the standards or libraries to abstract various embedded architectures.

A. Parallel Programming Model

There are a number of parallel programming models available in the general computing domain. Some of these are mature and widely adopted by the industries and academics. Such standard based models for the embedded domain are very scarce.

OpenCL[8] aims to serve as the data parallel programming model for heterogeneous platforms. But this is still too low level and performance portability is a big issue when reusing OpenCL programs for different architectures[9].

OpenMP[1] is the *de-facto* programming model in shared memory computation, and has been extended to heterogeneous architectures with the release of OpenMP 4.0 specification. Efforts to map OpenMP for embedded systems include: Our own previous work[6] that translates OpenMP to MRAPI and targets power processors. [10] implemented OpenMP on a high-performance DSP MPSoC, with efficient memory management and software cache coherence. What makes OpenMP a conducive model for embedded systems? The OpenMP *parallel for* construct could conduct data parallelism in an efficient manner, the *task* and *taskgroup* construct could allow programmers to easily explore task parallelism for irregular algorithms, and the newly added *target* construct brings big potential to support various types accelerators for offloading computation. However limited availability of resources, architectures of different types and programming for low-power restricts the adoption of OpenMP broadly on embedded systems. With the proposed solution framework in this paper, we could provide the embedded programmers an easy-to-adopt parallel programming model while ensuring a broad programming support for targeting embedded systems.

B. Standards for Programming Multicore Embedded Systems

Developing software products from vendors for embedded systems typically require implementations at the lower-level vendor-provided APIs level; thus making the learning curve steep while no software portability is guaranteed. The Multicore Association (MCA)[5] was founded by a group of leading semiconductor companies and universities, which provides a set of APIs to abstract the low-level details of embedded software development, and ease the efforts to resolve resource concurrency, communication, and task parallelism. Siemens recently developed an open-source implementation of MTAPI that we have used for our work in this paper to create an OpenMP-MTAPI prototype runtime library (RTL). This implementation is part of a larger project called *Embedded Multicore Building Blocks (EMBB)*[11] created to provide a better software solution for embedded systems. European Space Agency (ESA) builds their MTAPI implementation[12] for effortless configuration of SMP systems in their space products. Stefan et al.[13] provide a baseline implementation of MTAPI for their research on open tiled manycore SoC.

Heterogeneous System Architecture (HSA)[14] is a consortium lead by AMD and ARM. Unlike MCA APIs, the HSA expects a system level support for their specifications, for both hardware and software design.

3. Background

In this section, we briefly introduce OpenMP and MCA Task Management API (MTAPI) by discussing their essential features, usability and suitability for our work.

A. OpenMP

In this project, we map OpenMP task construct onto MTAPI. Therefore we will emphasize on OpenMP task construct in this subsection. OpenMP *task* and *taskwait* constructs were introduced in OpenMP 3.0 specification. They are updated with the definition of *taskgroup* and task dependency clauses in OpenMP 4.0 specification. OpenMP defines a task as a specific instance of execution that contains the executable codes and its data frame. The task constructs could be placed anywhere inside an OpenMP *parallel* construct, and explicit tasks are created when a thread encounters the *task* construct. Then these explicit tasks could be synchronized by using the *taskwait* construct. The tasks must pause and wait for their child tasks to be completed at the *taskwait* construct cause before moving on to the next stage in the program.

```
1  int fib(int n)
2  {
3    int i, j;
4    if (n<2) return n;
5    else{
6         #pragma omp task shared(i)
7         i=fib(n-1);
8         #pragma omp task shared(j)
9         j=fib(n-2);
10        #pragma omp taskwait
11        return i+j;
12   }
13 }
14 int main(void){
15   #pragma omp parallel shared(n)
16   {
17     #pragma omp single
18     printf ("fib(%d) = %d\n", n, fib(n));
19   }
20 }
```

Listing 1: OpenMP Task Example

The code snippet in Listing 1 illustrates the implementation of OpenMP task parallelism solution of Fibonacci numbers. OpenMP implementations rely heavily on threading libraries and operating system facilities. However, the embedded systems domain may lack such features. Some embedded systems do not even have an OS and are bare-metal posing more programming challenges. These systems have limited hardware resources so they cannot afford thread over subscription.

B. MCA Task API

The Multicore Association (MCA) defined the Multicore Task API (MTAPI) among several other APIs to abstract the lower level hardware details and provide a unique and easy-to-use API for different embedded architectures, thus expediting the software development procedures. MTAPI is designed to support both SMP and AMP systems, allowing an unlimited number of cores in the same or different architectures. Moreover, it allows implementation even on bare metal. The central concepts of MTAPI are listed as follows:

1) Domain and Node: MTAPI system is comprised of one or more MTAPI domains. An MTAPI domain is a unique system global entity. Each MTAPI domain consists of a set of MTAPI nodes. An MTAPI node is an independent unit of execution. A node can be a process, a thread, a thread pool, a processor, a hardware accelerator or an instance of an operating system. The mapping of MTAPI node is implementation-defined. In the MTAPI implementation we use for this work, an MTAPI node compromises a team of worker threads.

2) Job and Action: A job is an abstraction of the work to be done. An action is the implementation of a particular job. Different actions can implement the same job. An action could be a software or a hardware action. For instance, a job can be implemented by one action on a DSP board and another action on a general-purpose processor. The MTAPI API will pick a suitable action at the runtime when executing the job.

3) TASK: An MTAPI task consists of a piece of code together with the data to be processed. A task is light weighted with fine granularity; thus an application can create a large number of tasks. The tasks created within a node can be scheduled across different nodes internally. Each task is attached to a particular action at the runtime. The tasks are scheduled by the MTAPI runtime scheduler. The MTAPI runtime library, therefore, seeks an optimized way and a suitable action to handle the execution of the task. In this project, MTAPI tasks are created to handle the OpenMP explicit tasks.

4) QUEUE: The queue is introduced to guarantee the sequential order of task execution. The tasks associated with the same queue object must be executed in the order that they are attached to the queue. This feature is useful in the scenarios where the data must be processed in a specified order.

5) TASK GROUP: Task groups allow synchronization of a group of tasks. Tasks attached to the same group must be completed before the next step by calling 'group wait'. In this project, we map the OpenMP taskgroup construct seamlessly onto the MTAPI task group routines. We see that MTAPI has certain advantages to being the translation layer of our OpenMP RTL for the embedded systems domain. The MTAPI specification is designed for embedded systems.

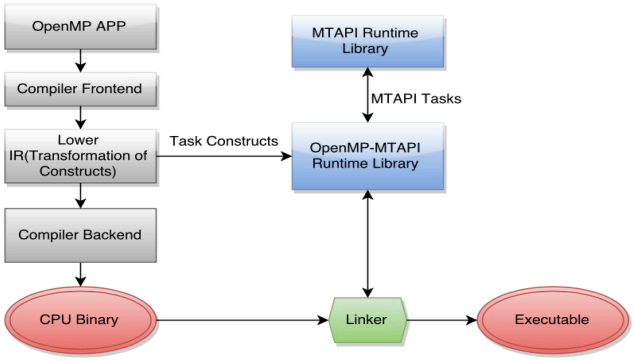

Fig. 2: OpenMP-MTAPI Solution Diagram

Unlike other parallelism programming models for the general-purpose computing systems, the MTAPI specification could be developed for resource-limited devices, heterogeneous systems which consist of different computation units with different ISAs, devices using embedded OSes or even bare metal. However, MTAPI is still a low-level library. A higher level programming model is desired to improve the programmers' productivity. Thus, it is necessary to introduce OpenMP to the parallel application development for embedded systems, with the lower-end support provided by MTAPI APIs.

4. Design and Implementation

A. Overall framework

We propose a comprehensive software stack for embedded systems by mapping the OpenMP constructs to MTAPI APIs. This work requires a deep learning curve of the OpenMP compilers and their runtime translations, as well as the environment configurations. In Figure 2, our framework shows an OpenMP compiler front-end translating the OpenMP constructs into OpenMP-MTAPI runtime function calls, therefore, the lower IR containing the transformation of the OpenMP application. The executable file is formed by linking OpenMP-MTAPI RTL and MTAPI RTL by the system linker. During the runtime of the application, the OpenMP-MTAPI RTL takes the OpenMP task function pointer and the task data pointer to create an explicit task. The OpenMP-MTAPI RTL then translates the explicit task onto MTAPI task, moves the control of execution onto the MTAPI RTL. After a parallel execution, the MTAPI RTL sends the results back to the OpenMP-MTAPI RTL, thus accomplishing the OpenMP task.

B. Mapping OpenMP runtime to MTAPI APIs

In this subsection, we give more implementation-level details.

1) Parallel Construct: In a conventional OpenMP RTL, when the master thread encounters an OpenMP parallel region, a set of worker threads will be created and be associated with a team. Thus, this is the "fork" of OpenMP's "fork-join" execution model. After the creation of worker threads, the OpenMP runtime will set them into the state of *wait* that could later be used to wake the worker threads up to a working state as and when needed. This way the overheads of the OpenMP

construct would be reduced since threads do not need to be created multiple times. In our enhanced OpenMP-MTAPI RTL, we intend not to handle threads directly; this is to ensure implementation portability. Instead, we use MTAPI solely to control the thread management and workload scheduling, to ensure the RTL's portability across different architectures and OSes. In our RTL, we initialize the default MTAPI node and create the default MTAPI action with the associated job handle when codes encounter the parallel region. Therefore, the program can start MTAPI tasks without further initialization later.

2) Task and Taskwait Constructs: OpenMP compilers translate each of the explicit tasks to an RTL function call with several parameters, including function pointer, data frame, arguments, and dependencies. Inside the RTL function, the tasks will be created and put into the task queue for further execution. For *taskwait* construct, OpenMP RTL will specify the descendant tasks of the current task, and wait for their completion before moving to the next step. We directly map OpenMP explicit tasks to MTAPI tasks in OpenMP-MTAPI RTL, by sending the function pointer and data frame to the previously created MTAPI action, and starting the corresponding task creations by calling *mtapi_task_create* routine, and storing the returned task handle for further reference of the tasks created. We also map the optional OpenMP group ID to the MTAPI tasks. In the context of MTAPI, *task wait* needs to utilize the task handle obtained from the creation of tasks and then waits for the completion of the corresponding task. Thus for the mapping of OpenMP tasks, we assign the task wait for the tasks created in the context and allow the completion of their child tasks, to meet the requirement of OpenMP taskwait construct.

3) Taskgroup Construct: OpenMP 4.0 specification introduced the *taskgroup* construct, which provides a simplified task synchronization mechanism. The taskgroup defines a structured region. All tasks created in the region including their descendant tasks belong to the same taskgroup. At the end of the taskgroup region, there is a synchronization point, thus forcing all tasks in this taskgroup to wait for completion before continuing execution. MTAPI specification provides a set of routines for taskgroup management. In the process of creating MTAPI tasks, programmers have options to whether or not to associate the tasks to a taskgroup. In our OpenMP runtime implementation, when encountering OpenMP taskgroup region, we create an MTAPI task group at the runtime with the default group ID and collect the returned group handle reference. Inside the group region while creating tasks, we specify which MTAPI taskgroup the tasks belonged to, by using the option to specify the taskgroup handle in the *mtapi_task_create* function. We also translate the OpenMP runtime call that refers to the end of taskgroup onto the *mtapi_group_wait_all*. This is performed with the taskgroup handle previously obtained when creating the task group, to wait for all the tasks and their child tasks in the group.

5. Performance Analysis

In this section, we evaluate the implementation of our OpenMP-MTAPI RTL. The experimental platform consists of two E5520 CPUs, with 16 threads available for execution, thus could provide sufficient room to explore the performance of OpenMP-MTAPI RTL over different numbers of threads. As mentioned earlier, we use the EMBB MTAPI implementation for our prototype OpenMP-MTAPI RTL.

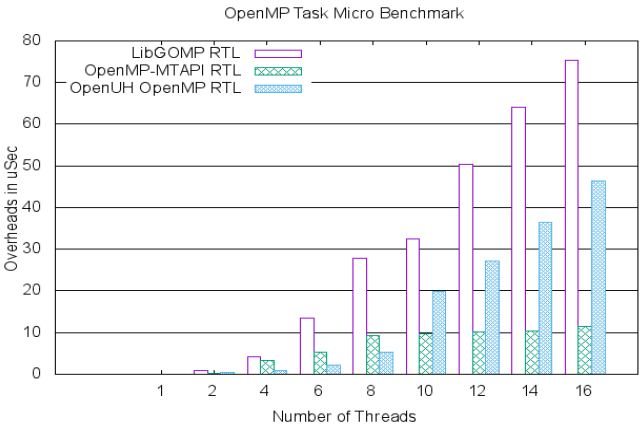

Fig. 3: OpenMP Task Overheads Measurement

A. OpenMP Task Overhead Measurement

To measure the performance of our translation, first up we find out if the addition of an MTAPI layer resulted in any overhead. We use the OpenMP task micro-benchmark suite [15] and compare the overheads of our OpenMP-MTAPI RTL against GCC OpenMP runtime library (LibGOMP RTL) and OpenUH-OpenMP RTL[16]. The overhead in this scenario is the difference between parallel execution time and the sequential execution time over an identical section of codes. Thus while testing, the micro-benchmark conducts a large number of iterations, for both task-parallel executions and sequential executions, therefore calculating the average differences and the overhead measurements. Figure 3 illustrates the comparison of the overheads. In the graph, the Y-axis indicates the actual overheads of the OpenMP task constructs in microseconds for threads ranging from 1 to 16. The OpenMP-MTAPI RTL performs similarly to the LibGOMP RTL for threads less than 4. However, we notice that the OpenMP-MTAPI RTL has significant advantages on a larger number of threads. The performance of OpenUH-OpenMP RTL has evidently fewer overheads for up to eight threads; however, when the number of threads increases to a larger amount, the OpenMP-MTAPI RTL achieves better performance. The results show that our prototype runtime library using OpenMP and MTAPI from Siemens incurred very less overhead. Minimum overheads for tasks creation and scheduling over a larger amount of threads is desired when MTAPI is deployed on multicore embedded devices.

B. Benchmark Analysis

In this subsection, we measure the performance of the OpenMP-MTAPI RTL using the BOTS suite[17]. We choose the Fibonacci and SparseLU for the analysis.

1) Fibonacci Benchmark: The Fibonacci benchmark uses a recursive task parallelization method to compute the nth Fibonacci number. This benchmark features small computation loads in tasks, but heavy dependency and synchronization

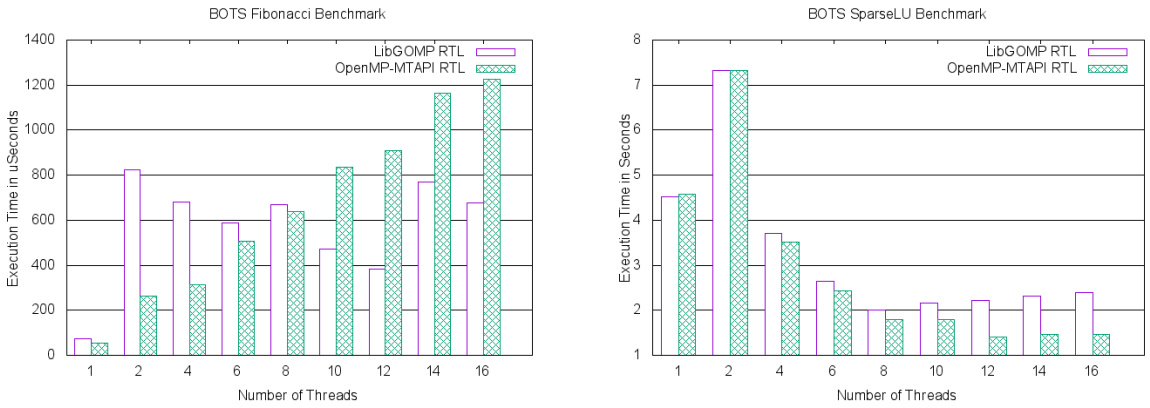

Fig. 4: Evaluating OpenMP-MTAPI RTL with BOTS benchmarks

among the tasks. As shown in Figure4(a), the OpenMP-MTAPI RTL performs better than LibGOMP RTL with less than eight threads while LibGOMP RTL has a distinct advantage with a larger number of threads. This shows our prototype implementation could be further improved, especially the synchronization strategies when using a lot of threads.

2) SparseLU Benchmark: The SparseLU benchmark computes an LU matrix factorization over sparse matrices. The matrix shows a lot of workload imbalance; using task parallelism and dynamic scheduling could lead to a good performance. In Figure4(b), we see that the OpenMP-MTAPI RTL achieves better performance when the number of threads is larger than eight while has similar performance with a lesser number of threads is being used. Unlike the Fibonacci benchmark, the SparseLU benchmark has a more substantial amount of computation workload per task. Thus, the time spent on synchronization has a smaller fraction of the total execution time. In this scenario, the efficiency of task scheduling and the overhead costs play a bigger role in performance.

6. Conclusion

In this paper, we design and create a portable, less-tedious, high-level and light-weight software solution for embedded platforms. The programmer does not need to rely on vendor-specific API. We translate OpenMP tasks and taskgroup constructs to MTAPI and create an OpenMP-MTAPI RTL. Evaluating our RTL, we demonstrate that no overhead was incurred and the results from OpenMP-MTAPI RTL is comparable to the GCC OpenMP RTL. We observed that the additional MTAPI layer did not incur any overheads to the overall solution stack. Using OpenMP-MTAPI RTL, an embedded programmer does not have to learn vendor-specific or low-level API but merely know to program using OpenMP. We will continue to explore the heterogeneous space and consider some specialized accelerators as part of the future work.

Acknowledgment

The authors of this paper would like to express our sincere gratitude to the anonymous reviewers. We would also like to thank Tobias Schuele and Urs Gleim from Siemens for their valuable feedback and suggestion along with Markus Levy, President of MCA and EEMBC for his continued support.

References

[1] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[2] "OpenMP application program interface version 4.0," http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013.

[3] E. Stotzer, A. Jayaraj, M. Ali, A. Friedmann, G. Mitra, A. P. Rendell, and I. Lintault, "Openmp on the low-power ti keystone ii arm/dsp system-on-chip," in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 114–127.

[4] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman, "Early experiences with the openmp accelerator model," in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 84–98.

[5] "Multicore Association Website," http://www.multicore-association.org.

[6] C. Wang, S. Chandrasekaran, P. Sun, B. Chapman, and J. Holt, "Portable mapping of openmp to multicore embedded systems using mca apis," in *ACM SIGPLAN Notices*, vol. 48, no. 5. ACM, 2013, pp. 153–162.

[7] P. Sun, S. Chandrasekaran, and B. Chapman, "Targeting heterogeneous socs using mcapi," in *TECHCON 2014, in the GRC Research Category Section 29.1*. SRC, September 2014.

[8] K. O. W. Group *et al.*, "The opencl specification," *version*, vol. 1, no. 29, p. 8, 2008.

[9] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of cuda and opencl," in *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 2011, pp. 216–225.

[10] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, "Implementing openmp on a high performance embedded multicore mpsoc," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–8.

[11] Siemens, "Embedded multicore building blocks from siemens," *URL https://github.com/siemens/embb*, 2014.

[12] D. Cederman, D. Hellström, J. Sherrill, G. Bloom, M. Patte, and M. Zulianello, "Rtems smp for leon3/leon4 multi-processor devices," *Data Systems In Aerospace*, 2014.

[13] S. Wallentowitz, P. Wagner, M. Tempelmeier, T. Wild, and A. Herkersdorf, "Open tiled manycore system-on-chip," *arXiv preprint arXiv:1304.5081*, 2013.

[14] P. Rogers and A. C. FELLOW, "Heterogeneous system architecture overview," in *Hot Chips*, 2013.

[15] J. LaGrone, A. Aribuki, and B. Chapman, "A set of microbenchmarks for measuring openmp task overheads," in *PDPTA*, vol. 2, 2011, pp. 594–600.

[16] C. Addison, J. LaGrone, L. Huang, and B. Chapman, "Openmp 3.0 tasking implementation in openuh," in *Open64 Workshop at CGO*, vol. 2009, 2009.

[17] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the

exploitation of task parallelism in openmp," in *ICPP'09*. IEEE, 2009, pp. 124–131.