On the Effectiveness of OpenMP teams for Programming Embedded Manycore Accelerators

Alessandro Capotondi
Università di Bologna
alessandro.capotondi@unibo.it

Andrea Marongiu
Swiss Federal Institute of Technology (ETH Zurich)
a.marongiu@iis.ee.ethz.ch

Abstract

With the introduction of more powerful and massively parallel embedded processors, embedded systems are becoming HPC capable. In particular heterogeneous on-chip systems (SoC) that couple a general-purpose host processor to a many-core accelerator are becoming more and more widespread, and provide tremendous peak performance/watt, well suited to execute HPC-class programs. The increased computation potential is however traded off for ease programming. Application developers are indeed required to manually deal with outlining code parts suitable for acceleration, parallelize there efficiently over many available cores, and orchestrate data transfers to/from the accelerator. In addition, since most many-cores are organized as a collection of *clusters*, featuring fast local communication but slow remote communication (i.e., to another cluster's local memory), the programmer should also take care of properly mapping the parallel computation so as to avoid poor data locality. OpenMP v4.0 introduces new constructs for computation offloading, as well as directives to deploy parallel computation in a cluster-aware manner. In this paper we assess the effectiveness of OpenMP v4.0 at exploiting the massive parallelism available in embedded heterogeneous SoCs, comparing to standard parallel loops over several computation-intensive applications from the linear algebra and image processing domains.

Categories and Subject Descriptors C.1.4 [*Computer Systems Organization*]: Processor Architectures—Parallel Architectures; D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; D.3.3 [*Software*]: Programming Languages—Language Constructs and Features

General Terms Performance assessment

Keywords Heterogeneous systems, manycores, HPC, NUMA, nested parallelism, OpenMP

1. Introduction

Architectural heterogeneity has proven an effective design paradigm to cope with an ever-increasing demand for computational power within tight energy budgets, virtually in every computing domain. Programmable many-core accelerators are nowadays widely used in high-performance computing systems [15] as well as in embedded devices, where they operate as co-processors under the control of a general-purpose CPU (usually called the *host*). Many-core accelerators are composed of several tens of simple processing elements (PEs), where highly-parallel computation kernels of an application can be offloaded to improve overall performance/watt. From the hardware design viewpoint, the most common approach is that of organizing the computation resources in *clusters*, each featuring a small-medium number of PEs tightly coupled to local L1 memory (typically designed as a scratchpad). This architectural template is no longer solely used in general-purpose graphical processing units (GPGPUs), but it is widespread also in embedded manycores [11] [5] [1] [4].

The tremendous GOps/Watt that such architectures can achieve are traded-off for an increased programming complexity: extensive and time-consuming rewrite of applications is required, using specialized programming paradigms. OpenCL [6], one of the most representative examples of such category of programming models, aims at providing a standardized way of programming such accelerators, however it offers a very low-level programming style. Higher-level programming styles are offered by directive-based approaches such as OpenACC [12] or OpenMP [13], which has included in the latest specification extensions to manage accelerators.

The main advantage of OpenMP-like approaches is that of enabling users to express software optimizations in a highly flexible and abstract manner, focusing on algorithmic details rather than architecture-specific aspects. Offloaded regions of codes are referred as target regions and may include sequences of sequential, parallel regions, and possibly nested parallel regions. Nested parallelism is particularly important to effectively use many cores organized as a fabric of *clusters*. Indeed, while communication to local L1 memory leverages fast and high-bandwidth channels such as crossbars, inter-cluster communication is subject to non-uniform memory access (NUMA) effects, as it relies on multi-hop transactions over a network-on-chip, which offers lower bandwidth and higher latency.

OpenMP 4.0 offers constructs to specify the distribution of work among clusters in a cluster-aware manner. Specifically, the `teams` construct allows the creation of a team of worker threads each belonging to a different cluster. Each team master can then create nested parallel teams, whose threads are recruited from local resources. In a scratchpad-based architecture, the master thread is typically responsible for bringing data in and out via DMA transfers, thus it is extremely important that the thread-to-core mapping follows a cluster-aware policy such as the one enabled by the `team` construct. Distributing work among threads in a locality aware manner can be done at the loop-level using the `distribute` clause.

In this paper we explore the benefits of using cluster-aware workload distribution in an embedded many-core accelerators, con-

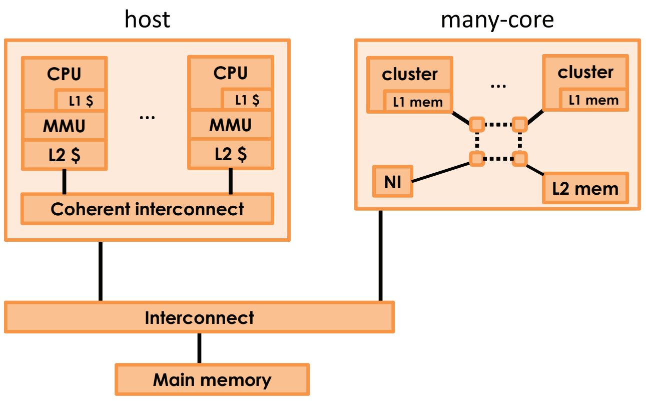


Figure 1. Heterogeneous embedded SoC template

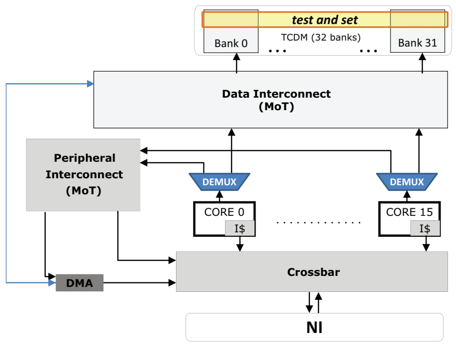


Figure 2. On-chip shared memory cluster

sidering several benchmarks from the linear algebra and image processing domain, and parallelizing them with OpenMP 4.0. We highlight the benefits of such recent additions to the specifications, comparing the results to a flat parallelization scheme, i.e., one which uses all the processors available from a single logical thread team. In this case only a single master thread for the whole platform is available to orchestrate data transfers, which generates computation with poor locality. We also compare the distributed approach to the use of standard OpenMP nested parallel regions and show that in absence of cluster awareness these perform even poorlier.

The rest of the paper is organized as follows. Section 2 describes the target heterogeneous SoC and cluster-based manycore. Section 3 discusses the key background notions for OpenMP v4.0. Section 4 introduces the considered benchmarks and discuss acceleration and parallelization schemes. Section 5 presents the evaluation of the described schemes. Section 6 describes related work and Section 7 concludes the paper.

2. Architectural Template

In this work we consider as a many-core based heterogeneous system the ST Microelectronics STHORM platform [11], but the results discussed later can be applied to a broader class of devices which share with STHORM a common architectural template. Figure 1 shows the block diagram of the target heterogeneous embedded system template. A powerful general-purpose processor (the *host*) is coupled to a programmable manycore accelerator composed of several tens of simple processors, where critical computation kernels of an application can be offloaded to improve overall performance/watt [1, 4, 5, 11].

Similar to GPGPUs, the many-core accelerator leverages a multi-cluster design to overcome scalability limitations [5, 11]. Processors within a cluster are tightly-coupled to local L1 scratchpad memory, which implies low-latency and high-bandwidth communication. Globally, the many-core accelerator leverages a partitioned global address space (PGAS). Every remote memory can be directly accessed by each processor, but inter-cluster communication travels through a NoC, and is subject to non-uniform memory access (NUMA) latency and bandwidth. Unlike the typical GPU data-parallel cores, that rely on a common fetch/decode phase, the processors considered here are simple independent RISC cores, perfectly suited to execute both single-instruction, multiple-data (SIMD) and multiple-instruction, multiple-data (MIMD) types of parallelism. This allows to efficiently support a programming model such as OpenMP, that leverages not only data-level parallelism, but also sophisticated forms of dynamic and irregular parallelism (e.g., tasking).

The simplified block diagram of the target *cluster* is shown in Figure 2. It contains sixteen RISC32 processor cores, each featuring a private instruction cache. Processors communicate through a multi-banked, multi-ported Tightly-Coupled Data Memory (TCDM). This shared L1 TCDM is implemented as explicitly managed SRAM banks (i.e., scratchpad memory), to which processors are interconnected through a low-latency, high-bandwidth data interconnect which allows 2-cycle L1 accesses (one for request, one for response). This is compatible with pipeline depth for load/store for most processors, hence it can be executed in TCDM without stalls – in absence of conflicts. The interconnection supports up to 16 concurrent processor-to-memory transactions within a single clock cycle, given that the target addresses belong to different banks (one port per bank). Multiple concurrent reads at the same address happen in the same clock cycle (broadcast). A real conflict takes place only when multiple processors try to access different addresses within the same bank. In this case the requests are sequentialized on the single bank port. To minimize the probability of conflicts i) the interconnection implements address interleaving at the word-level; ii) the number of banks is M times the number of cores (M=2 by default).

Processors can synchronize by means of standard read/write operations to an area of the TCDM which provides *test-and-set* semantics (a single atomic operation returns the content of the target memory location and updates it).

Since the L1 TCDM is typically very small (256KB for STHORM) it is impossible to permanently host all data therein or to host large data chunks. The software must thus explicitly orchestrate data transfers from main memory to L1, to ensure that the most frequently referenced data at any time are kept close to the processors. To allow for performance- and energy- efficient transfers, the cluster is equipped with a DMA engine.

The OpenMP v4.0 implementation that we consider for our exploration is based on our previous work [10] and has been extended to include all the features for kernel offloading.

3. Background

OpenMP v4.0 [13] introduces *offloading* directives to program accelerators. Similar to any previous OpenMP construct, these directives apply to the code block that they enclose. The key construct is the `target` directive, which highlights the structured code block that should be compiled and loaded for execution onto a device. The `map` clause can be additionally used to specify which data items have to be transferred to and from the device. In addition, the `target data` directive allows to allocate and transfer data before the actual offload takes place (i.e., a sort of data pre-fetching). The `device` clauses allows to specify the exact device to use if more than one is present in the system.

STRAS	Matrix multiplication using Strassen decomposition
GSID	Generalized squared interpoint distance
LRFR	Local reference frame radius (surface matching)
HIST	Histogram interpolation
NCC	Normalized cross-correlation algorithm
CT	Object tracking based on a specific color
FAST	Corner detector [16]

Table 1. Benchmarks

Within a `target` region most standard OpenMP constructs for parallelism can be used. Thus, upon offload a single thread is created that starts execution of the target region, until a `parallel` construct is encountered. Since many accelerators are organized into *clusters*, and since inter-cluster communication is typically costlier than internal transactions, OpenMP v.4.0 also introduces directives to abstractly expose architecture organization at the program level. The `teams` directive groups the threads of a device into sets (*teams*) that are later mapped onto physical clusters, thus achieving uniform and high-locality inter-thread communication. The programmer can control the number of teams (`num_teams` clause) and the maximum number of threads in each team (`thread_limit` clause) along with the teams directive, respectively. One of the threads in each team is designated team *master* and the structured block following the directive is executed by all team masters across the different teams. Upon team start only team masters execute, sequentially, one per cluster. When a `parallel` directive is encountered, all the threads in each team start execution, to collaborate in the execution of the enclosed structured block.

As most of the parallel work in offloaded kernels is typically found within loops, the `distribute` directive is provided to distribute loop iterations across teams, and then across threads therein. Note that the same thing could not be simply achieved by nesting two `parallel for` constructs, as this would require manually rewriting the loop as a nested loop (with outer and inner loops).

These new constructs allows to achieve a cluster-aware mapping of threads but also loops, without requiring that the programmer explicitly handles these aspects. In the next section we illustrate how these constructs can be used to efficiently offload computation to a many-core accelerator.

4. Benchmarks and Acceleration Schemes

In this section we briefly describe the six benchmarks used for our exploration, and the acceleration schemes enabled by the OpenMP v4.0 offload directives. The benchmarks were selected from the linear algebra, image processing and computer vision domains, and are representative of the computational kernels typically offloaded to many-core accelerators. A brief description can be found in Table 3.

FAST is particularly sensitive to input data, in terms of the available degree of parallelism. The two parameters that impact the performance the most are input image size and corner density. The former influences the overall number of iteration. Being the core computation kernel of FAST particularly fine-grained, a very small number of iterations per threads results in visible parallelization overheads. The latter influences the actual parallel work, which is protected by an if statement that quickly filters out image block that clearly don't contain a corner. For this reason, we consider here six variants of the benchmark execution, with as many different input images:

- **1.5%_S** 1.5% corner density in a small image (QVGA)
- **6%_S** 6% corner density in a small image
- **15%_S** 15% corner density in a small image
- **1.5%_L** 1.5% corner density in a large image (VGA)
- **6%_L** 6% corner density in a large image
- **15%_L** 15% corner density in a large image

Since in STHORM the host and the accelerator physically share the main L3 memory, the offload infrastructure by default simply passes pointers to data structures therein, rather than copying them to the accelerator space. However, for improved performance and energy efficiency, data has to be moved in the TCDM. In absence of a data cache this has to be explicitly done in the program via DMA transfers.

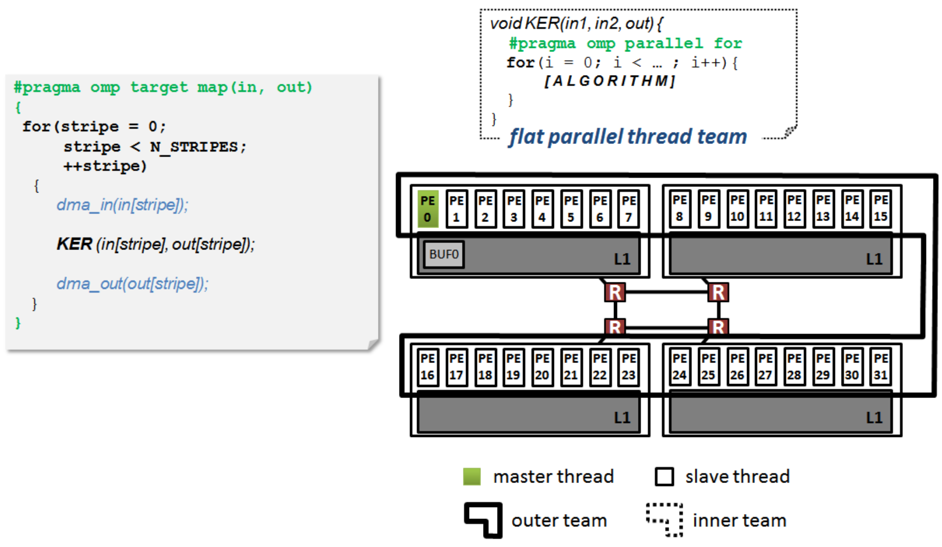
Each of the considered benchmarks operates on input and output data sets that are too large to fit in the TCDM. Thus, such data structures are divided in *stripes*, which are transferred in and out of the TCDM following a traditional double buffering scheme.

DMA transfers of data stripes are typically taken care of by a single thread (the *master*), from within an outer loop. Additional threads are involved in parallel computation when the transfer is complete. To parallelize the target benchmarks we have used three different approaches.

The simplest approach to use all available cores is that of creating a large parallel region which recruits them all. We call this approach *flat* parallelism, as it does not take into account the hierarchical structure of the cluster organization (interconnect, memory). Figure 3 shows how this parallelization scheme deploys threads onto available cores. The `target` directive starts execution of the enclosed region onto a single core, which orchestrates DMA transfers then jumps into the KER function. Here the main loop is found, and it is parallelized with a `parallel for` construct. By default, if no number of threads is specified all the available threads are involved. Note that since the master thread manages the DMA transfers with no awareness of the clusters, the data used by all threads in the parallel region is held in a single buffer (BUF0) that physically reside in the TCDM of the cluster that hosts the master thread. As a consequence, the threads that live in the same cluster as the master will enjoy fast data access, whereas threads belonging to other clusters will experience longer access times, leading to unbalanced computation.

Figure 4 shows the second parallelization approach, which adds awareness of the clustered nature of the platform to the code. Here the `teams` directive is used to create an outer parallel team that recruits threads from different clusters. These threads will become local masters of these clusters, and will orchestrate DMA transfers to/from the local TCDM. The `distribute` directive is used to partition the outermost loop among local masters, and this will make each master have its own data buffer in the local L1 memory. When a new `parallel` construct is encountered, an inner thread team is created, that shares high locality computation with the local master.

A third parallelization approach is considered for the sake of comparison: standard nested parallel regions. In principle is possible to specify the creation of an outer parallel region with as many threads as clusters, which will act as local master to those regions. Additional parallelism can then becreated when required by nest-



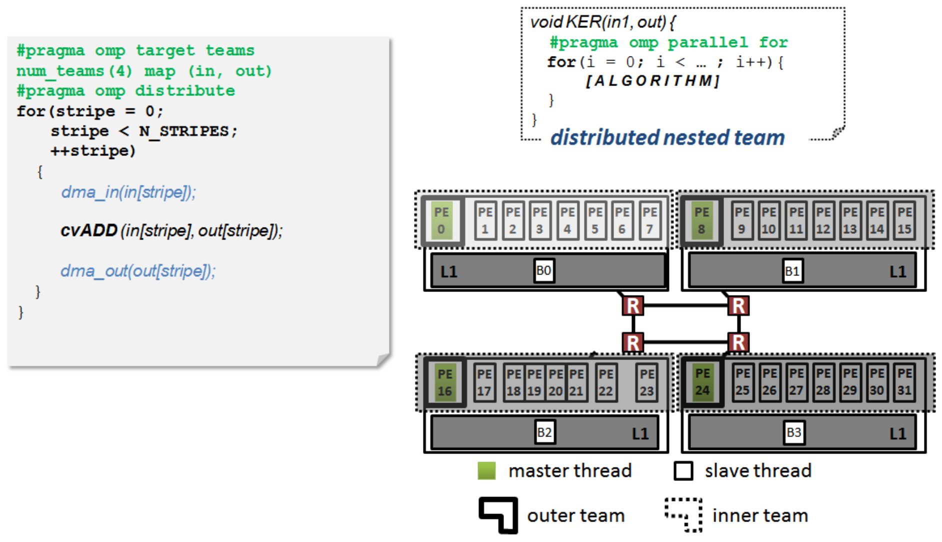
```
#pragma omp target map(in, out)
{
 for(stripe = 0;
      stripe < N_STRIPES;
      ++stripe)
  {
     dma_in(in[stripe]);

     KER (in[stripe], out[stripe]);

     dma_out(out[stripe]);
  }
}
```

```
void KER(in1, in2, out) {
  #pragma omp parallel for
  for(i = 0; i < … ; i++){
       [ALGORITHM]
  }
}
```

Figure 3. Flat parallelization of the Color Tracking application.



```
#pragma omp target teams
num_teams(4) map (in, out)
#pragma omp distribute
for(stripe = 0;
    stripe < N_STRIPES;
    ++stripe)
  {
     dma_in(in[stripe]);

     cvADD(in[stripe], out[stripe]);

     dma_out(out[stripe]);
  }
}
```

```
void KER(in1, out) {
  #pragma omp parallel for
  for(i = 0; i < … ; i++){
       [ALGORITHM]
  }
}
```

Figure 4. Distribute parallelization for color tracking.

ing a `parallel` construct within the first. Note that however this scheme lacks a notion of the cluster organization, and threads for the outer and inner regions will be recruited in an unspecified order. In the STHORM implementation this order is sequential, considering the list of all the processors available. Thus, creating an outermost region of four threads recruits the local masters from the same cluster. As a consequence, the code for DMA management will create four data buffers that reside in the same TCDM. Innermost teams will be composed of threads that physically belong to more than one cluster, which will create significantly higher cost for their runtime management (in addition to poor data locality). Figure 5 shows how this approach deploys threads and computation to the platform.

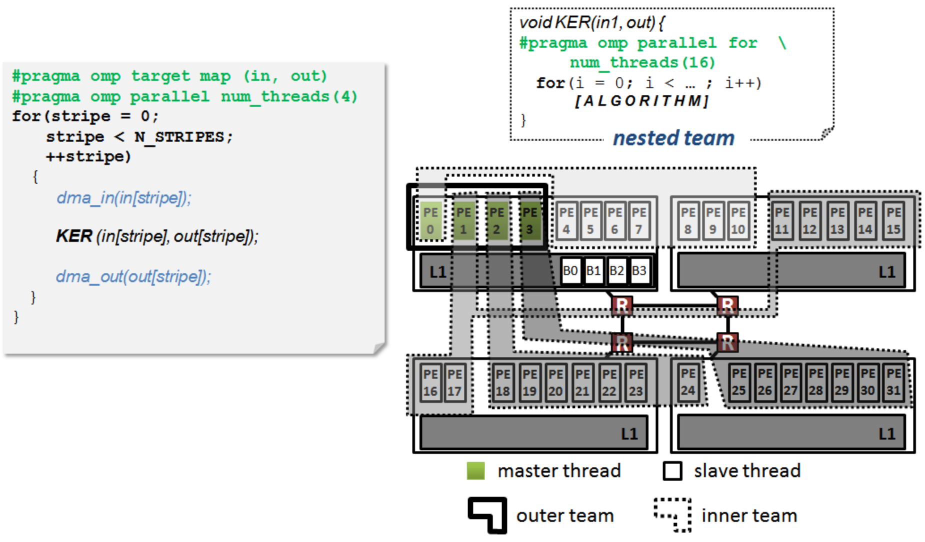
5. Experiments

In this section we describe the results collected by running the various benchmarks on STHORM when the three deployment approaches are considered. As a main metric of performance we consider speedup of the parallel application versus the sequential.

Results for this experiment are shown in Figure 6.

5.1 Effectiveness of the `teams distribute` construct

The most notable finding is that the cluster-aware workload deployment enabled by the `distribute` directive allows to achieve very high speedups and thus to make an effective use of many cores. Four out of seven benchmarks achieve nearly ideal speedup (above $60\times$), considering the best result for FAST speedup. As already explained, FAST leverages very fine-grained parallelization, for which the overhead introduced by runtime support for nested paral-



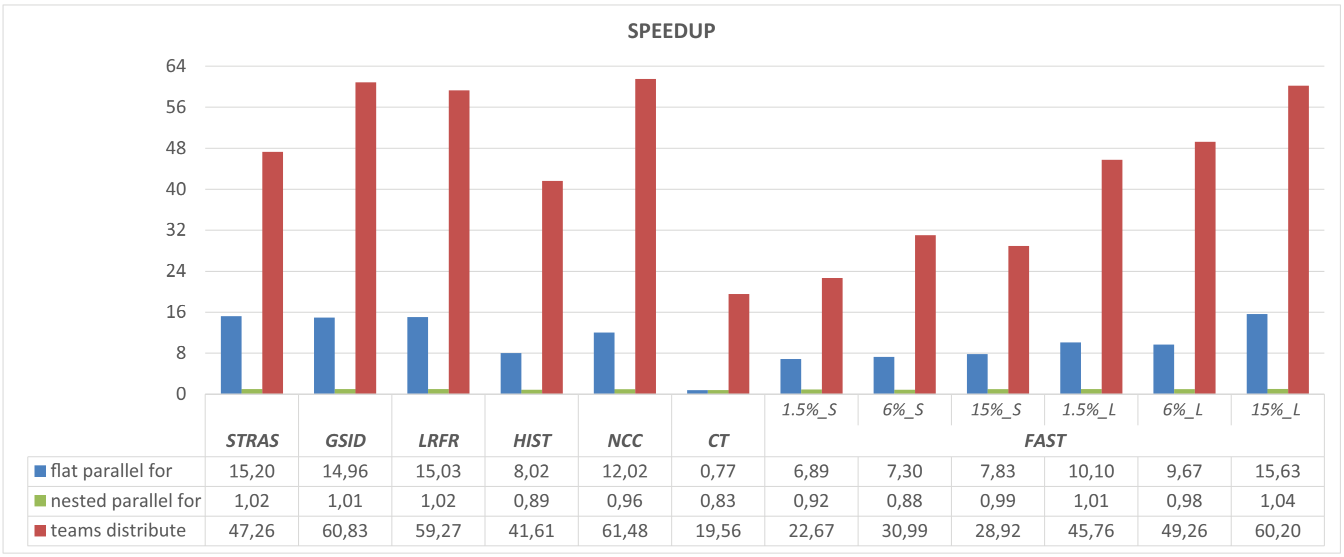
```
#pragma omp target map (in, out)
#pragma omp parallel num_threads(4)
for(stripe = 0;
    stripe < N_STRIPES;
    ++stripe)
  {
     dma_in(in[stripe]);

     KER (in[stripe], out[stripe]);

     dma_out(out[stripe]);
  }
}
```

```
void KER(in1, out) {
#pragma omp parallel for   \
       num_threads(16)
  for(i = 0; i < … ; i++)
       [ALGORITHM]
}
```

Figure 5. Nested parallel constructs on the Color Tracking application.



						FAST					
	STRAS	GSID	LRFR	HIST	NCC	CT	1.5%_S	6%_S	15%_S	1.5%_L	6%_L	15%_L
🟦 flat parallel for	15,20	14,96	15,03	8,02	12,02	0,77	6,89	7,30	7,83	10,10	9,67	15,63
🟩 nested parallel for	1,02	1,01	1,02	0,89	0,96	0,83	0,92	0,88	0,99	1,01	0,98	1,04
🟥 teams distribute	47,26	60,83	59,27	41,61	61,48	19,56	22,67	30,99	28,92	45,76	49,26	60,20

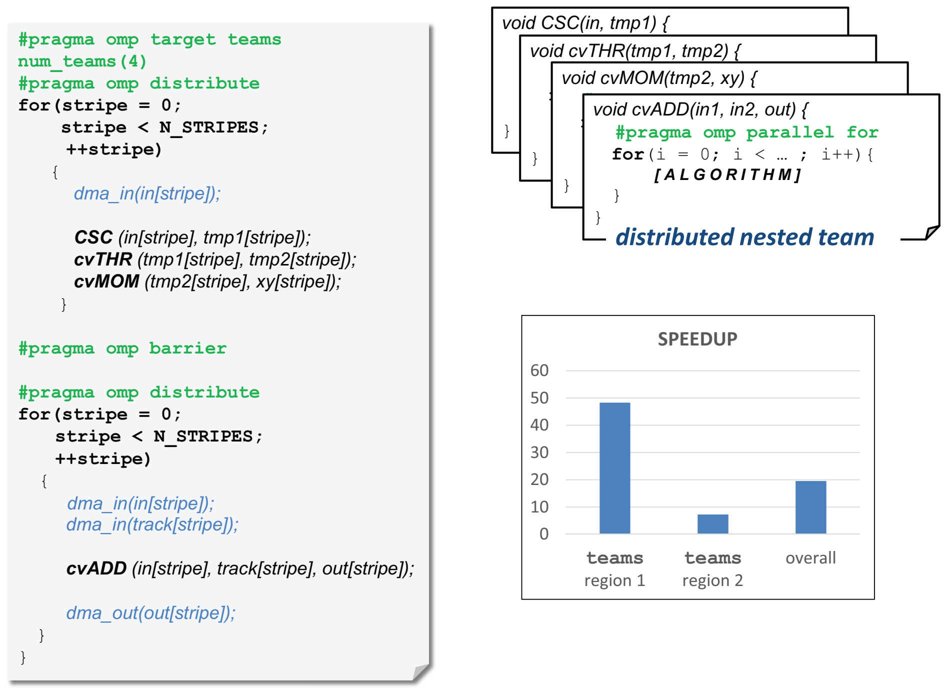
Figure 6. Comparison of various approaches to nested parallelism support.

lelism has a higher impact. Thus, when the image size is very small (QVGA) the speedups are limited (up to 60×) The corner density is also confirmed to have a big impact on performance, as shown by the variance among the three configurations (1.5%, 6%, 15%). Note that already for moderately large images (VGA) the speedups get as high as close to ideal.

The only application that achieves poor speedup in this configuration is CT, thus it is worth a bit more of investigation. Color-based tracking consists of a cascade of four functional kernels. Color space conversion (CSC), threshold-based color filter (cvTHR), motion vector calculation (cvMOM) and motion vector to reference frame addition (cvADD). Each of these kernels contains little computation, thus to improve the computation to communication ratio (CCR) we merge the CSC, cvThresh and cvMOM kernels into a single kernel (i.e., a single data stripe transfer is required to execute all the kernels in sequence). The last kernel, cvADD can not be merged with the previous kernels because it requires as an input the motion vectors for the whole image. Figure 7 illustrates the described parallelization scheme, with the first three kernels merged in a single `teams` region, plus a second `teams` regions composed of the sole last kernel. The figure also shows the breakdown of the speedup for these two teams regions. The CCR for cvADD is very small (only an addition is performed per pixel), and this justifies the small speedup achieved for this kernel, which overall impacts the total speedup for the application.

5.2 Comparison with flat `parallel for` construct

The comparison with the flat `parallel for` construct shows a much lower efficiency (speedup is always below 16×). As explained in Section 4 this is due to the poor locality of computa-



```
#pragma omp target teams
num_teams(4)
#pragma omp distribute
for(stripe = 0;
     stripe < N_STRIPES;
     ++stripe)
   {
     dma_in(in[stripe]);

     CSC (in[stripe], tmp1[stripe]);
     cvTHR (tmp1[stripe], tmp2[stripe]);
     cvMOM (tmp2[stripe], xy[stripe]);
    }

#pragma omp barrier

#pragma omp distribute
for(stripe = 0;
    stripe < N_STRIPES;
    ++stripe)
  {
     dma_in(in[stripe]);
     dma_in(track[stripe]);

     cvADD (in[stripe], track[stripe], out[stripe]);

     dma_out(out[stripe]);
  }
}
```

```
void CSC(in, tmp1) {
}
void cvTHR(tmp1, tmp2) {
}
void cvMOM(tmp2, xy) {
}
void cvADD(in1, in2, out) {
  #pragma omp parallel for
  for(i = 0; i < … ; i++){
     [ A L G O R I T H M ]
  }
}
```

Figure 7. Breakdown of CT kernels speedup.

tion generated by a deployment scheme which only envisions a global master for the entire manycore. This master will manage data stripes transfers into the local TCDM, but several threads from the same logical team reside on remote clusters. Such threads will have to traverse the NoC and compete with several other transactions, both for data requests coming from other threads and for instruction cache refills. It has to be pointed out that it is not only the actual parallel computation that encounters such remote communication issues. The implementation of the OpenMP runtime support also relies on data structures that are hosted in the TCDM of the cluster that hosts the master thread. Thus, every time that the parallel code requires explicit or implicit thread synchronization (e.g., barriers, end of parallelization constructs, dynamic loop scheduling, locks, etc.), additional remote transactions are generated. These results are even more important in the light of the fact that the non-expert programmer will always tend to use the flat parallel for approach as a default.

5.3 Comparison with nested `parallel for` construct

Probably the most surprising result is that achieved with the nested `parallel for` construct. Due to the above mentioned reasons regarding poor data locality and remote team management it was expected that the speedus would be limited. The extent to which this would impact performance could not entirely be expected. Nested parallel regions have traditionally been used in large HPC systems to improve the performance, however this was always done in combination with language or runtime constructs to control thread-to-core binding. Thus, while logically nested parallel regions and distributed teams are equivalent – in terms of how the work is split at the outermost level among *local masters*, and how innermost teams work in strict collaboration with these masters – physically the lack of control of where such masters and their slaves are mapped in the platform leads to extremely poor results. Note that, compared to the flat `parallel for` construct, in this case the impact of runtime library overhead is much more pronounced, as managing and synchronizing nested parallel teams generates much higher communication volume [9].

6. Related Work

The latest OpenMP 4.0 specifications introduce relevant features for accelerator exploitation, but not many devices are currently OpenMPv4-enabled. Among commercial devices Texas Instrument Keystone II [19] and Intel Xeon Phi [3] are probably the most representative examples. Stotzer [18] and Schmidl [17] present a performance assessment of flat parallelism for these architecture. These architecture, different from the embedded manycores considered in our work, rely on a coherent shared memory system and on multi-level data-cache hierarchy.

Bertolli et al. [2] propose a method to coordinate GPGPU threads mimicing the OpenMP 4.0 specification for Nvidia CUDA GPGPUs. [2] explores the utilization of the new `team` and `distribute` pragmas to implement efficiently dynamic parallelism on GPGPU accelerators. The focus of this work is however more on presenting a compiler implementation rather than assessing the effectiveness of the language constructs. Also Liao et al. [8] present an OpenMP 4.0 source to source compiler for Nvidia GPU. The compiler is based on the ROSE Compiler Infrastructure [7] and supports the OpenMP 4.0 `team` and `distribute` directives to deploy threads among CUDA cores. A more recent work from Yang et al. [20] presents a directive-based APIs *la* OpenMP that extends the CUDA language to enable dynamic nested parallelism and task level parallelism within a kernel. Ozen et al. [14] evaluate how different parallel programming interfaces, like OpenMP and other patterns for heterogeneous system can influence the deployment and the efficiency of kernel execution on GPGPUs in OmpSs. Unlike what is presented here, the focus for all these works in on GPGPU-like accelerator.

7. Conclusion

Manycore-based embedded heterogeneous SoCs are nowadays HPC capable, but efficiently programming them is a cumbersome task. OpenMP has always provided a user-friendly interface to application development, based on compiler directives that abstractly highlight parallelism in a sequential program. The latest specification version 4.0 introduces new constructs for computation offloading, as well as directives to deploy parallel computation with high data locality. This paper explores the capabilites of OpenMP v4.0 at exploiting the massive parallelism available in embedded heterogeneous SoCs. In particular, our experiments demonstrate that the new `teams distribute` construct allows to abstractly expose the clustered organization of most many-cores, thus achieving very efficient resource exploitation. Compared to standard parallel loops (the most widely used by inexpert programmers) with no awareness of the hierarchical interconnect and memory organization, these new construct enable major improvements in terms of speedup. Nested parallel loops, that logically provide a similar abstraction to the `teams distribute` construct, in absence of architectural awareness surprisingly prevent every speedup at all, in virtually every considered case.

Acknowledgments

This work was supported by EU project FP7 P-SOCRATES (611016).

References

[1] Adapteva. Epiphany III 16-core Chip Product. URL `http://adapteva.com/docs/e16g301_datasheet.pdf`.

[2] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave. Coordinating gpu threads for openmp 4.0 in llvm. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 12–21, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-7023-0. . URL `http://dx.doi.org/10.1109/LLVM-HPC.2014.10`.

[3] C. George. Knights corner, intels first many integrated core (MIC) architecture product. In *Hot Chips*.

[4] A. Heinecke, M. Klemm, and H.-J. Bungartz. From GPGPU to many-core: Nvidia Fermi and Intel Many Integrated Core architecture. *Computing in Science & Engineering*, 14(2):78–83, 2012.

[5] Kalray S.A. Kalray MPPA Manycore 256. URL `http://www.kalrayinc.com/kalray/products/#processors`.

[6] Khronos Group. The OpenCL specification. URL `http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`.

[7] Lawrence Livermore National Laboratory. ROSE Compiler Infrastructure. URL `http://rosecompiler.org/`.

[8] 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*, pages 84–98. Springer, 2013.

[9] A. Marongiu, P. Burgio, and L. Benini. Fast and lightweight support for nested parallelism on cluster-based embedded many-cores. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 105 –110, 2012.

[10] A. Marongiu, A. Capotondi, G. Tagliavini, and L. Benini. Simplifying many-core-based heterogeneous soc programming with offload directives. *Industrial Informatics, IEEE Transactions on*, 11(4):957–967, Aug 2015. ISSN 1551-3203. .

[11] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In *49th ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[12] OpenACC. The OpenACC Application Programming Interface. URL `http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf`.

[13] OpenMP ARB. OpenMP 4.0 application program interface. URL `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[14] G. Ozen, E. Ayguadé, and J. Labarta. On the roles of the programmer, the compiler and the runtime system when programming accelerators in openmp. In *Using and Improving OpenMP for Devices, Tasks, and More*, pages 215–229. Springer, 2014.

[15] Pete Decher. Embedding HPC: A rocket in your pocket. URL `http://www.embedded.com/design/prototyping-and-development/423`

[16] E. Rosten, R. Porter, and T. Drummond. Faster and better: a machine learning approach to corner detection. *IEEE transactions on pattern analysis and machine intelligence*, 32:105–19, 2010.

[17] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. S. Müller. Assessing the performance of openmp programs on the intel xeon phi. In *Euro-Par 2013 Parallel Processing*, pages 547–558. Springer, 2013.

[18] 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*, pages 114–127. Springer, 2013.

[19] Texas Instruments Inc. KeyStone II system-on-chip 66ak2hx. URL `http://www.ti.com/lit/ds/symlink/66ak2h12.pdf`.

[20] Y. Yang and H. Zhou. Cuda-np: Realizing nested thread-level parallelism in gpgpu applications. In *ACM SIGPLAN Notices*, volume 49, pages 93–106. ACM, 2014.