Targeting the Parallella

Spiros N. Agathos, Alexandros Papadogiannakis, and Vassilios V. Dimakopoulos(✉)

Department of Computer Science and Engineering, University of Ioannina, P.O. Box 1186, 45110 Ioannina, Greece
{sagathos,apapadog,dimako}@cse.uoi.gr

Abstract. Heterogeneous computing involves the combined use of processing elements with different architectures and is widely considered a prerequisite in the quest for higher performance and lower power consumption. To support this trend, the OpenMP standard has been recently augmented with directives that target systems consisting of general-purpose hosts and accelerator devices that may execute portion of a unified application code. In this work we present the first implementation of the OpenMP 4.0 accelerator directives for the Parallella board, a very popular credit-card sized multicore system consisting of a dual-core ARM host processor and a distinct 16-core Epiphany co-processor. We discuss in detail the necessary compiler and runtime infrastructures of our prototype, both for the host and the co-processor sides.

1 Introduction

Multicore processing units have become the dominant elements of modern computing systems. Personal workstations pack multiple compute cores in a socket, while high performance supercomputers combine general purpose multicore CPUs with specialized accelerator devices such as GPGPUs, DSPs and application-specific FPGAs. As a result, modern system architectures present a mix of different processor and memory hierarchies within the same system. At the same time the building blocks of such heterogeneous computing nodes are designed for different workload scenarios; multicore CPUs perform best in coarse grained tasks, while accelerators reach their computational potential in large scale data and fine grained vector processing.

The real challenge is to provide programming models that enable the extraction of satisfactory performance while also keeping programmer productivity at high levels in application development. Programming models such as OpenCL and CUDA [10] provide very efficient albeit rather primitive mechanisms for an application to exploit the hardware capabilities of GPGPUs and other devices. In addition, the heterogeneity of the system architecture leads to heterogeneous programming styles, requiring different code bases for the host CPU and the accelerators.

S.N. Agathos is supported by the Greek State Scholarships Foundation (IKY).

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 662–674, 2015.
DOI: 10.1007/978-3-662-48096-0_51

OpenMP, the de facto standard for shared-memory programming has been recently augmented with new directives that target arbitrary accelerator devices [17]. In the spirit of OpenACC [16], OpenMP 4.0 provides a higher level directive-based approach which allows the offloading of portions of the application code onto the processing elements of an attached accelerator, while the main part executes on the general-purpose host processor. What is important is that the application blends the host and the device code portions in a unified and seamless way, even if they refer to distinct address spaces.

The Parallella computer platform [5] is a recent and very popular credit card-sized multicore computer designed to be energy efficient and deliver high performance. It is an open source project and its processing power comes from a dual-core ARM CPU and a 16- or 64-core embedded accelerator, named Epiphany. The accelerator delivers up to 32 GFLOPS (102 GFLOPS, for the 64-core version) and is based on a 2D-mesh NoC of tiny, high performance, floating-point capable RISC cores with a shared global address space.

In this work we present the design and implementation of an OpenMP infrastructure for the Parallella board. It is the first OpenMP implementation for this particular system and also one of few OpenMP 4.0 implementations in general. We discuss both the compiler transformations and the runtime systems that provide the necessary support for the host and the device parts. Our implementation supports concurrent execution of multiple independent kernels. In addition it allows OpenMP directives within each offloaded kernel, supporting dynamic parallelism within the Epiphany.

The rest of the paper is organized as follows. In Sect. 1.1 we give an overview of related work. In Sect. 2 we present background material on the new OpenMP 4.0 device directives and summarize the Parallella board architecture along with its native programming models. We then describe our prototype implementation in detail in Sect. 3 while in Sect. 4 we present performance measurements. Section 5 concludes this work.

1.1 Related Work

Support for OpenMP 4.0 devices is fairly limited yet, both in the compiler side and the device side. In fact, the only commercial compiler that currently supports the `target` construct is the Intel ICC compiler and the only device it supports is the Xeon Phi [12]. Details of the offload procedure in the ICC compiler are given in [15].

Preliminary support for the OpenMP `target` construct is also available in the ROSE compiler. Chunhua et al. [13] discuss their experiences on implementing a prototype called HOMP on top of the ROSE compiler, which generates code for CUDA devices.

The GNU C Compiler has very recently added generic `target` support, designed to be tailored by device manufacturers, and combined with a runtime for the Intel Xeon Phi [1] accelerator. Bertolli et al. [7] propose a method to coordinate threads in an NVIDIA GPU using a single kernel as opposed to multiple kernels; they also discuss how their methods could be implemented as part of the LLVM compiler implementation of OpenMP 4.0. Finally, in [14] the authors present their implementation of OpenMP 4.0 on a TI Keystone II, where they use the DSP cores as devices to offload code to.

Regarding the Parallella board, higher-level parallel programming models are lacking. Offloading code to the Epiphany multicore chip is possible mainly through the native low-level eSDK [4] or using OpenCL as provided by the COPRTHR SDK [8]. The latter also provides a threading API similar to POSIX. Aaberge [3] analyzes the performance of Parallella and compares the two programming models, finding that generally the eSDK outperforms OpenCL. Finally, Varghese et al. [6] use the eSDK and raw assembly code to benchmark the Epiphany IV 64-core chip. They assess the effort required to extract good performance while noting the need for familiar, higher-level programming models.

2 Background

2.1 The OpenMP 4.0 Device Model

One of the goals of version 4.0 of the OpenMP API [17] is to provide a state of the art, platform-agnostic model for heterogeneous parallel programming. The extensions introduced since the previous version are designed to support multiple *devices* (for example accelerators, coprocessors, GPGPUs, etc.) without the need to create separate code bases for each device. The programmer simply marks portions of the (unified) source code to be offloaded to a particular device; the details of data and code allocations, mappings and movements are orchestrated by the compiler. The execution model is a host-centric one: program execution starts at the host processor (also considered a device) until one of the newly introduced constructs is met, which may cause the creation of data environment and the execution of a specified portion of code on a given device. The most important new directives are the *target*-related ones which mark the code and the data that are to be offloaded.

The `target` directive is used to transfer control flow to a device. The code in the associated structured block (kernel) is offloaded and executed directly on the device side, while the host task waits until the kernel finishes its execution. Each `target` directive may contain its own data environment which is initialized when the kernel starts and freed when the kernel ends its execution. In order to avoid repetitive creation and deletion of data environments, the `target data` directive allows the definition of a data environment which persists among successive kernel executions. Furthermore, the programmer can use the `target update` directive between successive kernel offloads to explicitly update the values of variables which are shared between the host and the device.

The memory for the data environment of a device is regarded as an autonomous extension of the OpenMP memory model. The data environment can be manipulated through `map` clauses within `target data` and `target` directives. These clauses determine how the specified variables are handled within the data environment. When an `alloc` map type is used an uninitialized variable is defined, whereas with a `to` map type the variable is additionally initialized from

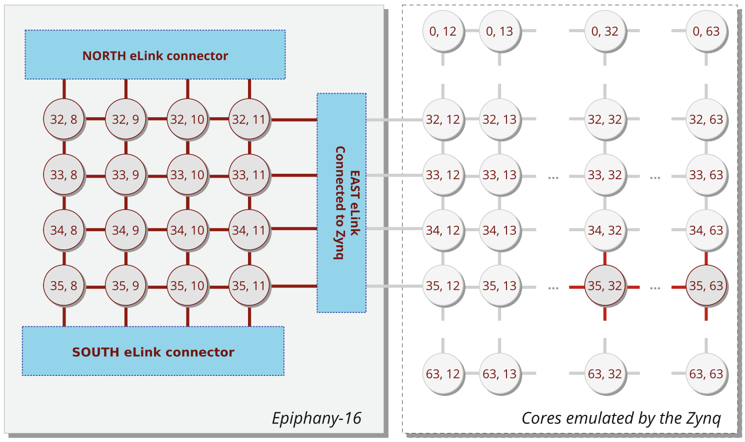


Fig. 1. The Epiphany mesh in a Parallella-16 board

the value of the corresponding host variable. If variable is mapped as `from` then an uninitialized device variable is defined; when the specified directive region finishes, the value of the device variable is copied back to the original host variable. If no type is specified or the type is `tofrom`, the variable is considered mapped as both `to` and `from`. Finally, the variables declared within `declare target` directives are also allocated in the global scope of the target device, and their lifetime equals the program execution time.

2.2 Parallella Board Overview

The Parallella-16 board [5] is an 18-core credit card sized computer and comes with standard peripheral ports such as USB, Ethernet, HDMI, GPIO, etc. The computational power of the $99 board comes from its two processing modules. The main (host) processor is a dual-core ARM Cortex A9 with 32 KiB L1 cache per core and 512KiB shared L2 cache, built within a Zynq 7010 or 7020 SoC. The other is an Epiphany 16-core chip which is used as a co-processor. The board has 1 GiB of DDR3 RAM, addressable by both the ARM CPU and the Epiphany. The former runs Linux OS and uses virtual addresses while the latter runs no OS and has a flat, unprotected memory map.

The Epiphany co-processor offers an impressive power efficiency that can reach up to 70 GFLOP/Watt, depending on the chip version. Two configurations of the Epiphany co-processor are currently available: the Epiphany-16 (with 16 cores and a 4×4 mesh NoC) and the Epiphany-64 (with 64 cores and an 8×8 mesh NoC). Although our discussion here holds for both versions, we refer mostly to the first one since it is widely available and is what our board contains. This particular chip is clocked at 600MHz and has a peak performance of approximately 25 GFLOPS (single-precision) with a maximum power dissipation of less than 2 Watt.

The architecture of the Epiphany is designed around a 64×64 mesh interconnect, so (in theory) systems up to 4096 Epiphany cores (eCOREs) are possible, by combining 16- and 64-core chips. On the Parallella-16 board, the Epiphany chip is pinned on a 4×4 submesh of the virtual 64×64 mesh whose north-west coordinates are (32, 8), as shown in Fig. 1. The chip has four eLinks (west, east, north and south), that may be used to interconnect it with other chips. In the current version of the Epiphany-16 chip the west eLink is inactive and the east eLink is connected to the Zynq host. Notice that the mesh NoC actually contains three separate meshes: the fast *cMesh* for writing on-chip memory, the *xMesh* for off-chip writes and the slowest *rMesh* for reading remote memory.

Each eCORE is a 32-bit superscalar RISC processor, capable of performing single-precision floating point operations, equipped with 32 KiB local scratchpad memory and two DMA engines. All eCOREs share a 32-bit address space with each one owning a 1MiB unique addressable slice; the scratchpad memory provides physically 32KiB of this slice. All memory is available through regular load/store instructions.

The Zynq, which is connected to the east eLink of the Epiphany, is perceived as the eastern part of the mesh. Based on the column-first routing scheme of the NoC, the Zynq can emulate the memory space of the cores in the 52 leftmost columns of the 64×64 virtual mesh, giving access to most of the board RAM to the Epiphany. A 32-MiB portion of the system RAM is left outside the Linux virtual memory manager area. From the Epiphany side it corresponds to the 32 cores located in coordinates from (35, 32) to (35, 63). This is designated as *shared memory* and is physically addressable by both the ARM and the Epiphany.

All common programming tools are available for the ARM host processor. For the Epiphany, the Epiphany Software Development Kit (eSDK) is available [4], which includes a C compiler and runtime libraries for both the host (eHAL) and the Epiphany (eLIB). A typical C program that utilizes the eSDK adheres to the following pattern: Initially the host executes some initializations and the sequential part of the application. Next, in order to offload code (kernel) to the co-processor it (a) initializes the Epiphany, (b) prepares the shared memory with all the data needed for the computation, (c) forms a workgroup of eCOREs and (d) triggers the execution of the kernel. All host-eCORE communication occurs through the shared memory.

3 Implementing OpenMP 4.0 on the Parallella

Our implementation is based on the OMPi OpenMP compiler [11]. OMPi is a lightweight OpenMP C infrastructure, composed of a source-to-source compiler and a flexible, modular runtime system. The input of the compiler consists of C code annotated with OpenMP pragmas and the output is an intermediate multithreaded code augmented with calls to the runtime system. A native compiler is used to generate the final executable. OMPi is an open source project that adheres to OpenMP V3.1 and targets general purpose SMPs and multicore platforms.

3.1 Compiling for the New Device Directives

The compiler has been extended to support the new OpenMP device model. In particular the input grammar has been modified to accommodate the new `target`-related directives. New nodes have been defined for the abstract syntax tree that represent the user code and new code generation routines have been introduced to produce the transformed code. The code generation phase now produces multiple output files, one for each different kernel (i.e. `target` region), plus the host code. The later contains the host part of the user program plus all kernels, since the host may be called to execute any of them, upon various runtime conditions. The kernel files are compiled using the eSDK tools.

When handling a `target data` directive, the compiler prepares a new data environment by injecting calls to the runtime system for each variable that appears in a `map` clause. The calls depend on the `map` type; specifically,

- For `alloc`-mapped variables, memory allocation calls are injected at the start of the construct block.
- For `to`-mapped variables, we additionally inject memory copy calls.
- `from`-mapped variables, are treated as `alloc` ones with additional calls to copy their values back to the original variable at the end of the construct block.
- For variables mapped as `tofrom`, we inject code as if the variable was both a `to`- and a `from`-mapped one.

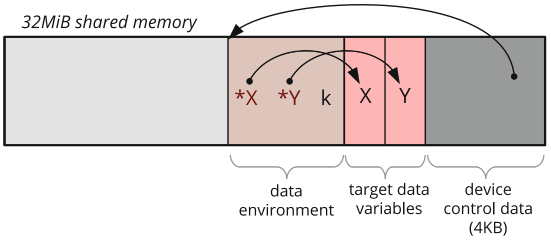
The above calls are preceded by a runtime call to mark the beginning of a new data environment; this is needed because the runtime system has to track the nesting of `target`/`target data` constructs for each device so as to activate the appropriate data environment when offload time comes.

The `target` construct is more complex because it behaves like a `target data` construct while in addition it offloads and executes code on the device by actually transferring both the code and the data environment to/from the device. For its transformation *outlining* is used, in a manner similar to the `parallel` and `task` constructs: the associated construct block is moved to a new function (kernel) which will serve as the offloaded kernel, with a single argument which points to the necessary data environment. In its place, a runtime call to offload the outlined kernel is placed.

Before the actual outlining of the construct takes place, the construct block is analyzed in order to discover any variables used in the code which were created outside of the construct (i.e. in parent `target data` regions). These, combined with the ones explicitly marked by `map` clauses, form the complete data environment of the kernel function. Depending on the type of mapping, variables in the data environment will be created as local copies of the original variable, initialized or not, or as pointers to the shared memory. Variables already existing in a parent data environment are replaced by pointers to their storage. For the Parallella, all such variables are stored in the shared memory area. For `alloc`-mapped variables we simply create a local variable with the same name within the kernel. We treat `to`-mapped variables in the same way we treat *firstprivate* variables in a `task` construct; a snapshot of the original variable is created by allocating

```
1 int X[10], Y[10];
2 int k;
3
4 #pragma omp target data map(X,Y)
5   #pragma omp target map(to:k)
6   {
7     /* Kernel code */
8   }
```

(a)



(b)

Fig. 2. Shared memory organization

space in the shared memory which is then initialized from the original variable. If the variable is of scalar type, a local variable is also defined within the kernel function and its value is copied from the shared memory in order to optimize access speed. No local copies are created for array types, due to the very stringent eCORE memory budget. The situation is similar for `from`-mapped variables. Here however, after the offload returns, the value is copied back from the shared memory to the original variable. Variables mapped as `tofrom` as well as variables which did not appear in any `map` clause are treated as if they appeared in a `map(to:)` clause with the extra copy-back steps of the `from`-mapped variables.

Finally, the `target update` directive is replaced by runtime calls to copy every variable in a `from` (or `to`) motion clause from (to) the shared memory to (from) the original host variable.

3.2 Runtime Architecture

At the host (Zynq) side the runtime system consists of two parts; the first is a full-fledged OpenMP runtime library, part of the regular OMPi infrastructure, necessary for supporting execution on the two ARM cores. The second part provides additional functionality, which is required for controlling and accessing the Epiphany device.

The communication between the Zynq and the eCOREs occurs through the shared memory portion of the system RAM as described earlier. The shared memory is divided in two sections, see Fig. 2(b). The first section is called Device Control Data (DCD) area, and it has a fixed size of 4KiB; it is used transparently by OMPi for kernel coordination and manipulation of parallel teams created within the Epiphany. The second part is used for storing the kernel data environments and part of the tasking infrastructure of the Epiphany OpenMP runtime described later. More specifically, during the preparation for offloading a kernel, a region is allocated to store the data environment of the kernel. This contains variables or pointers to variables which appeared in enclosing `target` or `target data` constructs and are not stored in the local memories of the eCOREs. An example is shown in Fig. 2(a). Variables `X` and `Y` in line 4 are annotated as `tofrom`. This causes a copy of each one to be created in the shared memory.

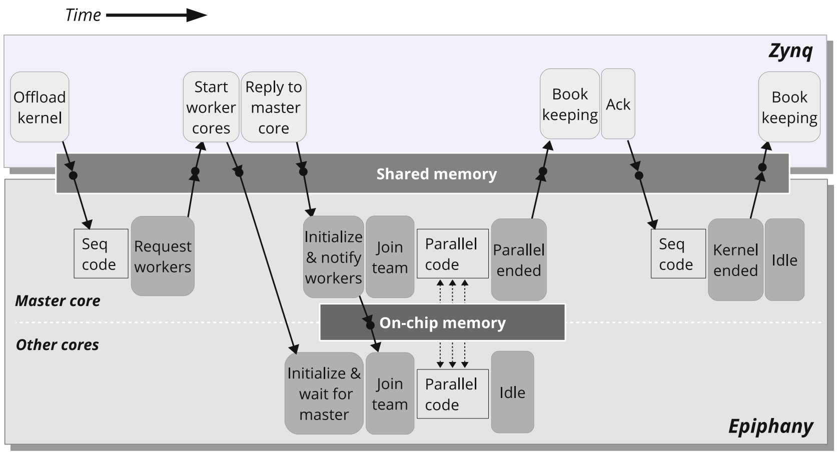


Fig. 3. Offloading a kernel containing dynamic parallelism

In line 5 the variable `k` is annotated as `to` and along with two pointers to `X` and `Y` form the data environment of the kernel. The beginning of the data environment is stored as a pointer in DCD, and is used by the kernel when starting its execution. All the above are stored at the higher end of the shared memory, leaving the lower end available for the programmer (e.g. for storing libraries which do not fit in the eCORE local memories).

In order to be able to control the eCOREs independently through eLIB calls, the initialization phase creates 16 workgroups, one for each of the available Epiphany's cores and puts them to the idle state for energy and thermal efficiency. For offloading a kernel, the first idle core is chosen and the precompiled object file is loaded to it for immediate execution. Because the current version of eHAL does not provide a way for an eCORE to notify directly the host for kernel completion, a special region of the DCD is designated to store special flags set by the eCOREs. The DCD infrastructure has a thread-safe design; this allows multiple host threads to offload multiple independent kernels concurrently onto the Epiphany.

3.3 OpenMP Within the Epiphany

The eCOREs do not execute any operating system and there is no provision for creating and handling dynamic parallelism (e.g. threads) within the Epiphany chip. In addition, the 32KiB local memory of each eCORE is quite limited, unable to handle sophisticated OpenMP runtime structures in addition to application data. As such, supporting OpenMP within the device side of the board is non-trivial.

The creation of a parallel team within an offloaded kernel is depicted graphically in Fig. 3. When a kernel is offloaded to a specific eCORE, the core executes its sequential part until a parallel region is encountered; the core will create a new team and become the master of the team. Because only the host can activate other Epiphany cores, the master core sends a request to the host through the device control data (DCD) section in shared memory, requesting the activation of a number of cores. The host-side thread which offloaded the kernel will activate as many cores as possible to satisfy the master request. A copy of the same kernel is then offloaded to the newly activated cores. The activated cores begin their execution by fetching all the appropriate information regarding the parallel team and its master core from the DCD section in shared memory. Immediately after that they spin waiting for the master to signal the execution of the parallel code. Once all required cores have been activated, the master has access to the actual team size and the coordinates of the team cores. A local flag is then set to release the team cores and let them execute the parallel region. During the parallel code execution all synchronization between the cores occurs through their fast local memories. When the region completes, the cores return to the idle, power saving state, while the master core informs the host thread about the termination of the parallel team. The host marks the idling cores as available for future use, and sends an acknowledgment to the master. The latter continues with the rest of kernel code.

We note that another, possibly faster, strategy for supporting dynamic parallelism would be to have all eCOREs loaded with the kernel(s) in advance and spin, waiting for the master to signal them which kernel to execute. However, this would increase power consumption dramatically and thus we did not pursue it further.

To support the OpenMP worksharing constructs (`single`, `for`, `sections`), the infrastructure originally designed for the host was trimmed down to a minimum so as to minimize its memory footprint; this is linked and offloaded with each kernel. The corresponding coordination among the participating eCOREs utilizes the structures stored in the local memory of the team's master core. This is possible because an eCORE can access any address in the Epiphany address space. In particular, while an eCORE may access its own scratchpad memory using *local* addresses (which range from 0_{16} to $7FFF_{16}$), its memory can also be globally accessed by all cores using its row and column coordinates: if r and c are the row and the column of a core, the start of its scratchpad memory is at address $r \times 4000000_{16} + c \times 100000_{16}$. The mesh coordinates of the master core are available to all team cores through the DCD area in shared memory.

The eSDK libraries for the Epiphany provide mechanisms for locks and barriers between the eCOREs. Their implementation is highly optimized to exploit the fast cMesh subnetwork as much as possible. Because they assume that the synchronized cores belong to the same workgroup, we modified them in order to adhere with our multiple cooperating workgroup organization. Additionally the barrier was augmented with task execution extensions. Our prototype tasking infrastructure is based on a blocking shared queue stored in the local memory of the master eCORE. The corresponding task data environments are stored in the shared memory.

Table 1. Size of empty kernel (bytes)

Scenario	OMPi	eSDK
1 kernel	7092	2232
16-core team	10560	3084

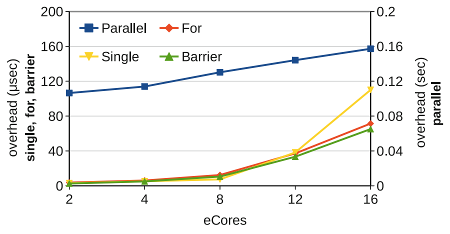


Fig. 4. Overhead results of EPCC benchmark

4 Measurements

We have conducted a number of tests in order to measure the efficiency of our offloading mechanisms alongside the space and timing performance of the OpenMP runtime within the Epiphany accelerator. Our board is the Parallella-16 SKUA101020 and we use eSDK 5.13.9.10. The system runs Ubuntu 14.04 with kernel 3.12.0 armv7l GNU/Linux. GCC and e-GCC v.4.8.2 were used as back-end compilers for OMPi.

4.1 Memory Footprint

To examine the memory overhead of our Epiphany runtime, which gets linked with each offloaded kernel, we created a set of simple OpenMP programs. The kernels were compiled with "-O3 -funroll-loops" flags and we used the *e-size* tool of the eSDK to examine the produced ELF object files. The results are shown in Table 1. In the first scenario, one effectively empty kernel is offloaded, containing only a single assignment. It can be seen that OMPi incures a 4.5KiB overhead as compared to an identical kernel created using the native eLIB. Examining the ELF, it is seen that our runtime requires approximately 1KiB more for its internal data and another 3.5KiB for its runtime routines. In the second scenario we create a team of 16 cores running the previous trivial kernel; for OMPi this is accomplished through a `parallel` directive while for the eSDK program we create a workgroup of 16 cores which are synchronized using a barrier. While the data section remains constant, the additional offloaded runtime routines cause an increase in the text section; approximately 7KiB more than the corresponding native kernel are required. Additional functionality is offloaded if the kernel contains worksharing constructs and this accounts for another 3KiB approximately. All in all, OMPi was found to require 4–10KiB more than a similarly structured eSDK-based kernel. While this is certainly non-negligible, we note that (a) our prototype has not been optimized yet, (b) some portions could be moved to shared memory as a tradeoff between local memory space and speed and (c) the programmability gains are rather significant.

4.2 Overheads

The EPCC micro-benchmarks suite [9] is widely used to measure OpenMP construct overheads for a particular implementation. In order to measure OMPi overheads within the Epiphany, we created a modified version of the benchmarks. Their basic routines are offloaded through `target` directives and executed as kernels without further modifications. Measurements are taken from the host side, after subtracting any offloading costs. In Fig. 4 we present a sample of the results regarding the overheads of `parallel`, `for`, `single` and `barrier` constructs. The results are quite satisfactory, in all but the `parallel` construct. This is explained in part, because as described in Sect. 3.2, the formation of a dynamic team of cores incurs significant host-device communication, which includes additional kernel offloads. However, it should be stressed that offloading even an empty kernel has an overhead of at least 0.1 s, needed for resetting the core(s) that will execute it. Eliminating this cost, would require keeping all eCOREs active all the time, sacrificing power efficiency.

4.3 Mandelbrot Application

We tested OMPi using a simple version of the Mandelbrot deep zoom application which calculates a Mandelbrot set and zooms in and out up to 10500$\times$ at six predefined points. The whole frame by frame image is written directly to the frame buffer of the Parallella board (with a resolution of 1024×768), resulting in an impressive colorful video. The full traversal generates 204 frames per zoom point. The code for this application is one of the examples included with the eSDK in order to exhibit the real time performance possibilities of the Epiphany chip. Initially a host thread activates all 16 cores to execute the computation kernel. The kernel itself distributes the work statically among the cores; each core calculates the colors for a region of the image and writes the values to the frame buffer. At the end of each frame, all cores inform the host thread and wait to be synchronized. When all cores finish their caculations for the particular frame the host signals them to continue with the next one.

In order to utilize OpenMP, we unified the host and Epiphany code in a single file, moving the kernel code into a `target` region. Next, we removed all calls to eSDK and replaced them with OpenMP pragmas, and finally we removed the synchronization code, since this functionality is now carried out by a `barrier`. The generated kernel size was 11794 bytes; the original kernel was 4728 bytes, in comparison. The execution results are shown in Table 2. We give the total number of frames and the frame rate (i.e. the total number of frames divided by the execution time) for the original application and the OpenMP-based version. For comparison we also provide results of the application when the Zynq is used as the device that executes the kernels. In any given column, the differences between the frame rates is natural because of the variability of pixel calculations (darker pixels incur fewer computations).

As it can be easily seen, the original eSDK application performs from 8 % to 13 % better than the OpenMP-based one. We consider this as a very small

Table 2. Frames per second for the Mandelbrot deep zoom application (1024 × 768)

#frames	eSDK@Epiphany	OMPi@Epiphany	OMPi@Zynq
204	17.854	15.829	4.139
408	15.250	13.630	3.469
612	13.411	12.292	3.015
816	12.528	11.632	2.794
1020	13.330	12.304	2.997
1224	14.486	13.234	3.288

difference, given that our prototype is not yet highly optimized. Moreover, the OpenMP version, without any further modifications resulted in a total of 198 program lines, while the original required 301 lines of code. What is more important is that the programmability gains are huge. We achieved on average 90 % of the performance of the original application with a mere 5 OpenMP pragmas. Finally, notice that the Epiphany achieves up to 4× more frames per second as compared to the Zynq.

5 Conclusion and Future Work

We presented the design of the first OpenMP 4.0 infrastructure for the Parallella board. Our system treats the Epiphany-16 as an accelerator device, attached to a dual-core ARM host processor and allows the dynamic creation of parallel teams within the device itself. While not highly optimized yet, our prototype is able to support OpenMP 4.0 applications delivering performance up to 92 % of hand-written low-level eSDK code as observed for a particular application.

Currently, our prototype has a number of limitations which have to do with the handling of OpenMP internal control variables (ICVs) which are mostly lacking for the Epiphany. Another limitation is the lack of sophisticated management for the shared memory in the host runtime. The memory segments defined for kernel and tasking data environments are relinquished in the order they were allocated, which may cause unnecessary fragmentation. We are currently working on an improved allocator.

Our future work is concentrated mostly on two areas; first, optimize the current implementation and second, implement additional OpenMP functionality. For the former, we are working on minimizing both the memory footprint of the device runtime as well as its overheads for the OpenMP constructs. For the latter, our next target is the support of the new `teams` and `distribute` directives, which create a given number of thread teams within the accelerator, and divide loop iterations among them.

Acknowledgment. The authors would like to thank Adapteva for providing them with a Parallella-16 board through the Parallella University Program.

References

1. GCC 5 Release Series. https://www.gcc.gnu.org/gcc-5/changes.html
2. OpenMP/Clang. http://www.clang-omp.github.io/
3. Aaberge, T.: Analyzing the Performance of the Epiphany Processor. Master's thesis, Norwegian Univ. of Science and Technology, Aug 2014
4. Adapteva: Epiphany SDK reference Manual, Sept 2013
5. Adapteva: Parallella Reference Manual, Sept 2014
6. Varghese, A., Bob Edwards, G.M., Rendell, A.P.: Programming the adapteva epiphany 64-core network-on-chip coprocessor. In: Proceedings of the IPDPSW 2014, pp. 984–992. Phoenix, USA, Dec 2014
7. Chow, E., Anzt, H., Dongarra, J.: Asynchronous iterative algorithm for computing incomplete factorizations on GPUs. In: Kunkel, J.M., Ludwig, T. (eds.) ISC High Performance 2015. LNCS, vol. 9137, pp. 1–16. Springer, Heidelberg (2015)
8. Brown Deer Technology, LLC: COPRTHR API Reference (2014)
9. Bull, J.M.: Measuring Synchronisation and scheduling overheads in OpenMP. In: Proceedings of the 1st EWOMP, Lund, Sweden, pp. 99–105, Sept 1999
10. Kirk, D.B., Hwu, W.-M.W.: Programming Massively Parallel Processors. A Hands-on Approach, 2nd edn. Morgan Kaufmann, MA (2012)
11. Dimakopoulos, V.V., Leontiadis, E., Tzoumas, G.: A portable C compiler for OpenMP V. 2.0. In: Proceedings of the EWOMP 2003, Aachen, Germany, pp. 5–11, Sept 2003
12. Intel Corporation: User and Reference Guide for the Intel C++ Compiler 15.0, OpenMP* Support. https://software.intel.com/en-us/node/522679
13. Liao, C., Yan, Y., de Supinski, B.R., Quinlan, D.J., Chapman, B.: Early experiences with the OpenMP accelerator model. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 84–98. Springer, Heidelberg (2013)
14. Mitra, G., Stotzer, E., Jayaraj, A., Rendell, A.P.: Implementation and optimization of the OpenMP accelerator model for the TI keystone II architecture. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 202–214. Springer, Heidelberg (2014)
15. Newburn, C.J., Deodhar, R., Dmitriev, S., Murty, R., Narayanaswamy, R., Wiegert, J., Chinchilla, F., McGuire, R.: Offload compiler runtime for the Intel® Xeon Phi^{TM} Coprocessor. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2013. LNCS, vol. 7905, pp. 239–254. Springer, Heidelberg (2013)
16. OpenACC: The OpenACC Application Programming Interface Vesion 2.0, June 2013
17. OpenMP A.R.B.: OpenMP Application Program Interface V4.0, July 2013