OpenMP 4.0 Device Support in the OMPi Compiler

Alexandros Papadogiannakis *Spiros N. Agathos*[*]
Vassilios V. Dimakopoulos
Department of Computer Science and Engineering, University of Ioannina
P.O. Box 1186, Ioannina, Greece, GR-45110
{apapadog,sagathos,dimako}@cse.uoi.gr

Abstract

OpenMP 4.0 represents a major upgrade in the language specifications of the standard. Important constructs for the exploitation of SIMD parallelism, the support for dependencies among tasks and the ability to cancel the operations of a team of threads have been added. What is arguably the most important addition, however, is the introduction of the *device* model. A variety of computational units, such as GPUs, DSPs and general or special purpose accelerators are viewed as attached devices, where portion of a unified application code can be offloaded for execution. In this work we present the infrastructure for device support in the OMPi research compiler, one of the few compilers that currently implement the new device directives. We discuss the necessary compiler transformations and the general runtime organization. For the first time, special emphasis is placed on the important problem of data environment handling. In addition, we present a prototype implementation on the popular Parallella board which exploits the dual-core ARM host processor and the 16-core Epiphany accelerator of the system.

1 Introduction

OpenMP, the de facto standard for shared-memory programming, has been recently augmented with new directives that target arbitrary accelerator devices [18]. In the spirit of OpenACC [17], OpenMP 4.0 provides a higher level directive-based approach; it 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. In contrast, programming models such as OpenCL [13] and CUDA [14] provide efficient but rather primitive mechanisms for an application to exploit the hardware capabilities of GPGPUs and other devices. Such models project the heterogeneity of hardware directly onto software, forcing different programming styles and multiple code modules to accommodate each of the devices that is to be utilized.

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

Modern architectures present a mix of different processor organizations and memory hierarchies within the same system. Systems as small as a personal workstation may pack many compute cores in a socket, and/or combine general purpose multicore CPUs with accelerator devices such as GPGPUs, DSPs and specialized, application-specific FPGAs. Given the multiplicity of devices and the diversity of their architectures, the real challenge is to provide programming models that allow the programmer to extract satisfactory performance while also keeping his/her productivity at high levels. OpenMP 4.0 strives to play a major role in this direction, letting the programmer blend the host and the device code portions in a unified and seamless way.

Although support for the OpenMP 4.0 device model has been slow to be adopted by both compiler and device vendors, it is gaining momentum. Currently, the Intel ICC compiler [12, 16] and GNU C Compiler, GCC (as of the latest version [11]) support offloading directives, with both of them only targeting Intel Xeon Phi as a device. GCC offers a general infrastructure to be tailored and supplemented by device manufacturers. Preliminary support for the OpenMP `target` construct is also available in the ROSE compiler. Chunhua et al. [8] discuss their experiences on implementing a prototype called HOMP on top of the ROSE compiler, which generates code for CUDA devices. A discussion about an implementation of OpenMP 4.0 for the LLVM compiler is given by Bertolli et al. [5] who also propose an efficient method to coordinate threads within an NVIDIA GPU. Finally, in [15] the authors present an implementation on the TI Keystone II, where the DSP cores are used as devices to offload code to.

In this work we present an infrastructure for device support in the context of the OMPi OpenMP compiler [9], currently one of the few compilers that implement the OpenMP 4.0 directives for offloading code onto a device. We discuss the necessary compiler transformations and the general runtime organization, emphasizing the important problem of data environment handling. To the best of our knowledge, this is the first time this problem is given detailed consideration in the open literature; we present novel runtime structures to address it efficiently. While we deal mostly with the device-agnostic parts of the infrastructure, we also discuss our experiences with a concrete implementation on the highly popular Parallella board [2]. It is the first OpenMP implementation for this particular system, and supports the concurrent execution of multiple independent kernels. In addition, it allows OpenMP directives in each offloaded kernel, supporting dynamic parallelism within the system coprocessor.

The paper is organized as follows. In Section 2 we give the necessary background material. We present the compiler transformations in Section 3, while the corresponding runtime issues are discussed in Section 4. We then describe our prototype implementation for the Parallella board in Section 5. Finally, Section 6 concludes this work.

2 Background

One of the goals of version 4.0 of the OpenMP API [18] was to provide a state of the art, platform-agnostic model for heterogeneous parallel programming by extending its widely accepted shared-memory paradigm. The extensions introduced are designed to support multiple *devices* (accelerators, coprocessors, GPGPUs, etc.) without the need to create separate code bases for each device. Portions of the unified source code are simply marked by the programmer for offloading 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: execution starts at the host processor, which is also considered a device, until one of the new constructs is met; this may cause the creation of a data environment and the execution of a specified portion of code on a given device.

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, 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 may use the `target update` directive between successive kernel offloads to explicitly update the values of variables shared between the host and the device.

The execution of an OpenMP program has a set of initial device data environments (that is, a set of variables associated with a given code region), one for each available device. 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. 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.

The original and the corresponding variables, have common name, type and size but the task that executes in the context of a device data environment, refers to the corresponding variable instead of the original one. Data environments can be nested and a corresponding variable of a device data environment is inherited by all enclosed data environments. This means that a variable can not be re-mapped during the definition of nested data environments.

3 Compiler Transformations

The OMPi compiler [9] is a lightweight OpenMP C infrastructure, composed of a source-to-source compiler and a flexible, modular runtime system. The input of the compiler is C code annotated with OpenMP `pragma`s 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 and targets general-purpose SMPs and multicore platforms. It adheres to V3.1 of the OpenMP specifications, but support for V4.0 is under way. In addition to the device constructs presented here, initial implementations exist for the cancellation constructs, the `taskgroup` construct and task dependencies.

The main transformation step of OMPi for `parallel`, `task` and `target` directives is *outlining*. A new function is created, containing the transformed body, and the construct is replaced by a runtime call with the new function and a struct as parameters. The struct contains any variables declared before the construct but used in the body of the construct, and is initialized according to the data-sharing attribute clauses. In the following sections, we will use the code in Fig. 1 as an example in order to illustrate the transformation details.

When outlining a `target` region, we store a copy of the outlined function along with any other outlined functions that may occur during the transformation of its body (e.g. when having a `parallel` region inside the `target`), in a global list. After the main transformation phase of the code, we use the information stored in that list to produce kernel files, one for each `target` construct.

3.1 Target Data

According to the specifications, if a variable that appears in a `map` clause, already exists in the device data environment, no new space should be allocated and no assignments should occur. Since the `device` clause is an arbitrary expression and there is no restriction on the `device` clause of nested `target data` directives, the compiler is unaware of the devices that a variable has already been mapped on. For example, there could be a `target data` that adds variable a in device 1 and then another `target data` with the same variable that does not contain a `device` clause, and therefore will use the default device which, however, can be changed during execution.

To overcome this subtle problem we inject calls to the runtime for creating, initializing and finalizing environments with variables that appear in `map` clauses, regardless of whether they have appeared or not in an enclosing `target data` directive, and let the runtime handle it, as described in Section 4.1. In Fig. 1, this occurs for variables `x` and `y` in lines 12 and 13.

When transforming the `target data` directive, a `start` and an `end` call are injected before and after the body of the directive. If the directive is nested in another `target data` directive, the latter data environment is passed in the `start` call (Fig. 1, line 10). For each variable, depending on the map type, an `alloc` (for alloc/from) or an `init` (for to/tofrom) call is inserted at the start of the body (Fig. 1, lines 12–13). If the variable is from/tofrom a `finalize` call is inserted at the end of the body (Fig. 1, lines 28–29).[1]

3.2 Target

Each `target` directive is outlined similarly to `parallel` and `task` directives, albeit with different handlers for the variables. We create a new data environment, as if the construct was a `target data` one (lines 8–13 in Fig. 1). All variables that have already appeared in any enclosing data environment are inserted in the new one. Pointers to these variables are then placed in the *devdata*-struct, which will be passed to the outlined function. The pointers are initialized using runtime calls to get the address of the variables on the device space (lines 20–21). As an optimization, if a variable does not appear in any enclosing `target data` directive, space for the variable is created directly within the *devdata*-struct, instead of using a pointer (line 22).

[1]Similar calls are used when transforming `target update` directives.

Original code:

```
#pragma omp target data map(x,y)
   #pragma omp target map(from: x) device(2)
      x=y=z=1;
```

Transformed code:

```
 1 {                              // start target data
 2    _devid = -1;                // default device
 3    _ddenv = _start_ddenv(NULL, _devid, ..);
 4    _initvar(&x, sizeof(x), _ddenv, _devid);
 5    _initvar(&y, ..);
 6
 7    {                           // start target
 8       _devid = 2;              // requested device
 9       _ddenv_prev = _ddenv;
10       _ddenv = _start_ddenv(_ddenv_prev, _devid, ..);
11
12       _allocvar(&x, ..);       // ignored if default device is 2
13       _initvar(&y, ..);        // ditto
14
15       struct __dd__ {
16          int (* x);
17          int (* y);
18          int z;
19       } * _devdata = _devdata_alloc(_devid, sizeof(struct __dd__));
20       _devdata->x = get_vaddress(&x, ..); // request address @device
21       _devdata->y = get_vaddress(&y, ..);
22       _devdata->z = z;         // optimized
23
24       ort_offload_kernel(_kernelFunc0_, _devdata, ..); // kernel code
25
26       z = _devdata->z;
27
28       _finvar(&x, _ddenv);
29       _finvar(&y, _ddenv);
30
31       _end_ddenv(_ddenv);
32    }                           // end target
33
34    _finvar(&x, _ddenv);
35    _finvar(&y, _ddenv);
36    _end_ddenv(_ddenv);
37 }                              // end target data
```

Figure 1: Compiler transformation example.

In our example, the kernel body (`x=y=z=1;`) has been moved to an outlined function `_kernelFunc0_()`; the actual execution of the kernel occurs in line 24, where the runtime call is given the function name and the *devdata*-struct.

3.3 Declare Target

When the compiler encounters a `declare target` region, it stores any contained functions, function prototypes and variables in lists, and also marks them in the symbol table. The body of the directive is left as is during the main transformation phase of the code. Any declared variables within the `declare target` region are ignored during the transformation of the `target` directive.

3.3.1 Host Code.

The transformation for the `declare target` constructs occurs after the normal transformation phase ends, where all the declared variables have already been marked. A new static function is created in every source file for registering the declared variables with the runtime. For each initialized variable, a separate static variable is also created, using the same initializer. This is needed by the runtime when the initialization of the declared variable takes place on the available devices.

The above function is called whenever a `target update` or a `target` directive is met, which uses one of the `declare`d variables. This guarantees that the runtime registers all these variables before they are actually used. In addition, precautions are taken so that the operation is completed only once and is not subject to concurrent invocations. Finally, the `declare`d variables, which are used in a `target` region, are placed in a separate struct and are given to the runtime at offload time; in the example of Fig. 1, this struct would be an additional argument to the offloading call in line 24.

3.3.2 Kernel Code.

For each `target` directive we produce a separate kernel file. The code of a kernel starts with the `declare`d function prototypes. All variables declared in `declare target` regions are converted to pointers. Then, we transform the declared functions and the outlined function of the `target` directive, replacing any occurrences of the declared variables by pointers.

Moreover, a wrapper function is created, which is the first function called by the runtime library of the device. The wrapper serves two purposes; first it initializes the pointers for the declared variables from the struct passed in the offload function, and second, it calls the actual outlined kernel function.

4 Runtime Support

The runtime system of OMPi has been extended to support part of the new features of OpenMP 4.0. Except for the new runtime functions and environmental variables, the existing worksharing infrastructure was updated to support the cancellation directives. In addition, the tasking infrastructure adds preliminary support for the `taskgroup` functionality, as well as for task dependencies.

Regarding the new device model, OMPi is organized as a collection of modules which implement support for accelerators. The core module which coordinates OpenMP execution on the host is a largely unmodified version of the V3.1 runtime of OMPi; it is only augmented with functionality needed for the coordination of devices, and with a device interface that acts as glue between the host and the other device modules. On the other hand, new device modules are created, with each of them responsible for the manipulation of a particular device. Each module is divided into two parts; the first part is executed on the host and the second is executed on the the device, accompanying the offloaded kernels.

```
 1  int a, b;
 2
 3  #pragma omp target data map(a)        // DE1
 4    #pragma omp parallel num_threads(2)
 5    {
 6      int c;
 7      #pragma omp target data map(b)    // DE2^(1), DE2^(2)
 8        #pragma omp target map(a, c)   // DE3^(1), DE3^(2)
 9          ....
10    }
```

Figure 2: Nested device data environments created by a team of threads.

4.1 Data Environment Handling

The host device needs a bookkeeping mechanism in order to store and retrieve information regarding the variables that constitute the data environments. This information is used when the host accesses these variables for reading or writing, for example during the mapping of a variable or before/after the execution of a kernel, when the `target update` directive is used. This information is also used during the initialization phase of a kernel. Because of the arbitrary nesting of data environments, and the possibility of multiple devices, the bookkeeping cannot be statically handled at compile time. In what follows, we present our runtime solution to this non-trivial problem.

To allow the host to have fast access to information regarding the variables involved in a data environment, we utilize a special mechanism based on typical separately-chained hash tables (HT). It works approximately as a functional-style compiler symbol table [4], albeit operated by the runtime. A sequence of nested data environments in the source program produces a dynamic sequence of HTs with entries for the mapped variables. The information stored on each entry includes the id of the device which the mapping refers to, and a pointer to the actual storage space of the variable. The hash function takes as input the original variable address combined with the device number and returns the corresponding bucket. Collisions are handled through separate chaining.

We present our mechanism through two illustrative examples; detailed analysis follows right after. In Fig. 2 we show a code snippet where in line 3 a data environment (DE1) is created with the mapping of variable a. Then a team of two threads is created on the host device and each thread defines a separate data environment (DE$2^{(1)}$ and DE$2^{(2)}$) which includes the variables a and b. Finally, each thread offloads a code block, while at the same time creates a new data environment (DE$3^{(1)}$ and DE$3^{(2)}$) which includes the variables a and c. All the mappings in the example refer to the default device.

The sequence of the HTs for the above example is given in Fig. 3. The result of line 3 is the creation of DE1 which stores only one entry, that for variable a. The creation of the thread team does not affect the bookkeeping mechanism, and both threads can access the data of DE1 without the need of locks. Line 7 defines DE$2^{(1)}$ and DE$2^{(2)}$ (one for each thread); this causes the creation of new HTs, both of which are created as copies of the previous HT. As a result, each thread retains access to the enclosing

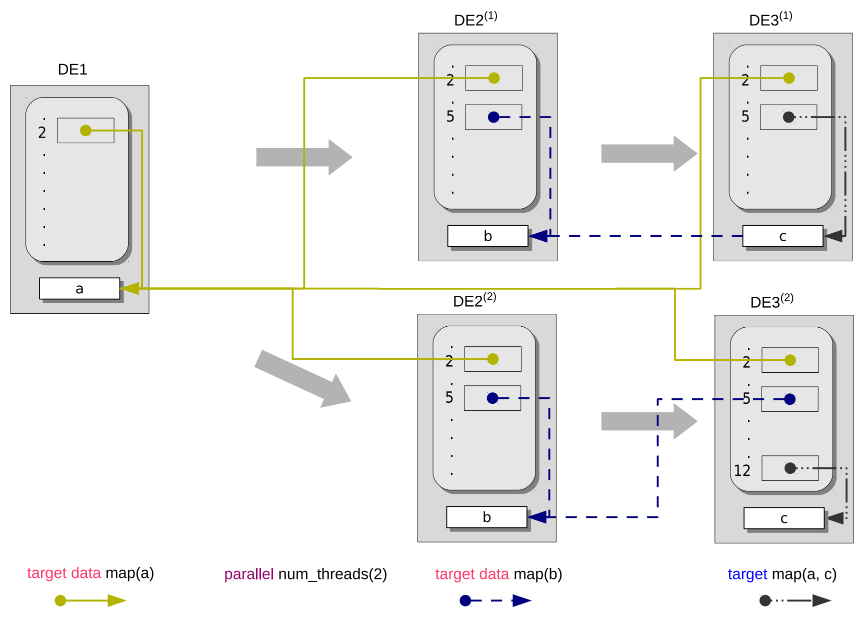

Figure 3: Hash table sequence for code in Fig.2. Solid yellow, dashed blue and mixed black arrows denote allocations and definitions made at lines 3, 7 and 8, respectively.

```
1  int d, e, f;
2
3  #pragma omp target data map(d) device(1)        // DE4
4    #pragma omp target data map(d, e) device(2)   // DE5
5      #pragma omp target map(f) device(1)        // DE6
6      {
7        e = d++; // implicit mapping of variable e
8        ...      // in device 1
9      }
```

Figure 4: Nested device data environments created for two different devices.

data environment. The mapping of variable b adds a new entry for DE2$^{(1)}$ and DE2$^{(2)}$ and the addition of a pointer from the HT to this entry. Similarly, in line 8 we have the creation of new HTs that handle DE3$^{(1)}$ and DE3$^{(2)}$, respectively, as copies of the previous two HTs. In the case of the second thread, variable c is hashed onto bucket 12, which is a free bucket. In contrast, variable c of the first thread caused a collision (hashed onto bucket 5), resulting in a chain between the variables c and b.

An additional complication arises by the possible presence of multiple devices, where a data environment may be nested within another environment that refers to a different device, as can be seen in the code of Fig. 4. In this example, in line 3, a data environment is defined for device 1 (DE4), with variable d. Then, DE5 is created in line 4 for device 2 that includes variables d and e. Finally, DE6 is created in line

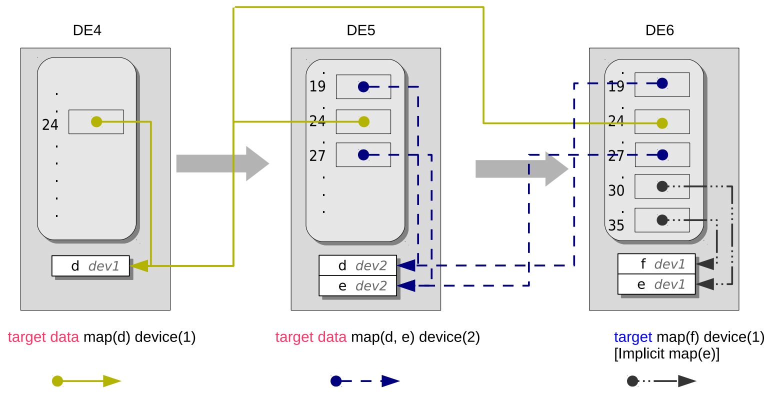

Figure 5: Hash table sequence for code in Fig.4. Solid yellow, dashed blue and mixed black arrows denote allocations and definitions made at lines 3, 4 and 5, respectively.

5 for device 1 and a block of code is offloaded for execution. Notice that d is already mapped through DE4 while there is an implicit mapping for variable e on device 1. The corresponding sequence of HTs is shown in Fig. 5. In DE4 there is only one entry for d on device 1. The HT for DE5 starts with a copy of DE4; entries for variables d and e are then added in buckets 19 and 27, correspondingly. Notice that because d now refers to device 2, the hash function results to a different bucket than the one used for device 1. Finally, DE6 starts as a copy of DE5 and has two entries added for variables f and e. Since d is already present in the data environment of device 1, no further actions are required.

4.1.1 Details of the Mechanism.

The data handling mechanism is operated by the host, resides in the host address space, and is independent from any attached devices. The HTs allow for efficient variable insertion and look-up operations. Each time a nested data environment is created, a new HT is initialized as a copy of the HT used by the enclosing data environment, as in functional-style symbol tables; destruction of the data environment requires a single memory deallocation for the corresponding HT. If a team of threads is created within a `target data` region, separate HTs are created, when needed, for each thread; this completely eliminates mutual exclusion problems.

In addition, we employ a further optimization. When initializing a new data environment for a particular device, the compiler informs the runtime about the *maximum possible number of variables added* to the environment. This is calculated statically during the analysis of the program; an exact number is not possible to derive because the devices the environments refer to are given by full expressions and can only be calculated at runtime, in the general case. In practice, only variables not already mapped on this device are actually inserted in the HT. Because of this information, the runtime is able to acquire memory for the HT *and* the entries in a single allocation request (this explains why in Fig. 3 and 5 the HTs and the corresponding entries lie within the same rectangle). This also has the desirable side-effect of increased data locality, as the hash table and the entry information reside on the same memory block.

The space requirements of the presented mechanism have an upper bound of $O(L \cdot K + n)$, where L stands for the maximum number of alive data environments, K is the size of a hash table (derived from the static compiler information discussed above) and n is the total number of variables that are mapped in all data environment definitions. In a program that defines a total of E data environments, exactly $\Theta(E)$ of memory allocations and deallocations are made, which include the memory required for the HT and the memory used for the entries. With a load factor at most equal to 1, the average time for an insertion or a lookup is $\Theta(1)$.

5 The Epiphany Accelerator as a Device

The Parallella-16 board [2] is a popular 18-core credit card-sized computer and comes with standard peripheral ports such as USB, Ethernet, HDMI, GPIO, etc. The computational power of the board comes from its two processing modules. The main (host) processor is a dual-core ARM Cortex A9 with 512 KiB 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, whereas the latter does not have an OS and uses a flat, unprotected memory map.

Two versions of the Epiphany co-processor are actually 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). Although our discussion here holds for both versions, we refer mostly to the first one since it is the one widely available. Each Epiphany core (eCORE) is a 32-bit superscalar RISC CPU, clocked at 600 MHz, capable of performing single-precision floating point operations, and equipped with 32 KiB local scratchpad memory and two DMA engines. The ARM and the Epiphany use a 32 MiB portion of the system RAM as *shared memory* which is physically addressable by both of them. All common programming tools are available for the ARM host processor. For the Epiphany, a Software Development Kit (eSDK [1]) is available, which includes a C compiler and runtime libraries for both the host (eHAL) and the Epiphany (eLIB). Furthermore, OpenCL is provided by the COPRTHR SDK [6]. The latter also provides a threading API similar to POSIX.

5.1 Runtime Organization

In this section we will briefly describe the key features of the runtime module that implements the support of the Epiphany accelerator as an OpenMP 4.0 device, based on the eSDK. More details are available in [3]. The module consists of two parts; one executed by the host CPU and one executed by the eCOREs.

5.1.1 The Host Part.

The communication between the Zynq and the eCOREs occurs through the shared memory portion of the system RAM. The shared memory is logically divided in two sections: The first section is a fixed size of 4 KiB, and 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 library.

In order to be able to control the eCOREs independently through eHAL calls, the initialization phase creates 16 workgroups, one for each of the available Epiphany 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 kernel object file is loaded to it for immediate execution. Due to the high overheads of the eSDK when offloading kernels to different workgroups, we developed an optimized low-level offload routine to assist the creation of OpenMP teams. We support multiple, independent kernels, executing concurrently within the Epiphany. 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 shared memory is designated for synchronization with the eCOREs.

For the `target teams` directive special care is taken regarding the choice of the eCOREs which will execute the associated kernel. In order to keep intra-team NoC traffic localized, the team masters are placed as if a `spread` thread affinity policy was in effect for them. For example, if all eCOREs are idle and the creation of 4 teams is requested, then the eCOREs with ids 0, 3, 12 and 15 (the ones in the four corners on the 4×4 grid) will be activated.

5.1.2 OpenMP within the Epiphany.

Supporting OpenMP within the device side presents many challenges due to the lack of dynamic parallelism and the limited local memory of each eCORE. Regarding the former, when a kernel is offloaded to a specific eCORE, the core executes its sequential part until a `parallel` region is encountered. Because only the host can activate other eCOREs, the master core contacts the host, requesting the activation of a number of cores. A copy of the same kernel is then offloaded to the newly activated cores. 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 small scratchpad memory makes it impossible to fit sophisticated OpenMP runtime structures alongside the application data. To support the worksharing constructs of the OpenMP, the infrastructure originally designed for the host was trimmed down so as to minimize its memory footprint; this is linked and offloaded with each kernel. The coordination among the participating eCOREs utilizes structures stored in the local memory of the team master. The eSDK provides mechanisms for locks and barriers between the eCOREs but they assume that the synchronized cores belong to same workgroup. Since in our runtime each eCORE constitutes a different workgroup, we were forced to modify all these mechanisms. Finally, our tasking infrastructure is

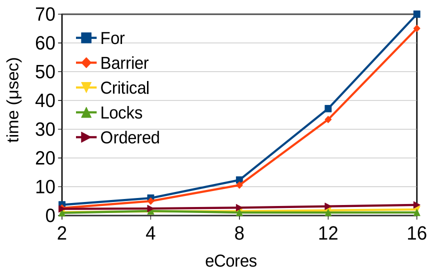

Figure 6: EPCC synchronization results

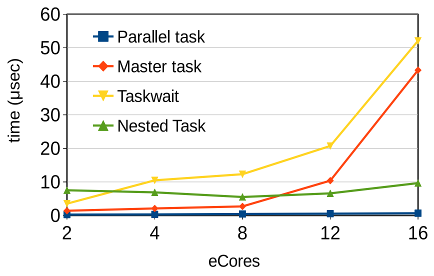

Figure 7: EPCC tasking results

based on a blocking shared queue stored in the local memory of the master eCORE, while the corresponding task data environments are stored in the shared memory.

5.2 Experiments

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. To examine the memory overhead of our Epiphany-resident runtime, we created a set of simple OpenMP programs to compare with the size of the kernels produced when the native eLIB is used. In the case of an empty kernel, containing only a single assignment OMPi incurs a 4.5 KiB overhead as compared to the kernel created by the native eLIB. In other scenarios which involved a team of 16 cores and complex OpenMP functionality, up to 10 KiB more than a similar eLIB-based kernel were needed. We are currently optimizing the runtime memory footprint even more.

In order to measure the OpenMP construct overheads within the Epiphany, we created a modified version of the EPCC microbenchmark suite [7] where their basic routines are offloaded through `target` directives. In Fig. 6 we plot a sample of the synchronization benchmark results. While most results are quite satisfactory, our initial prototype employs a non-optimized barrier, which also has a direct impact on the overheads of the `for` construct. We are currently optimizing its behavior. Sample results for the tasking benchmark are given in Fig. 7. The noticeable cases are those of the Taskwait and Master task tests. The contention on our simple lock-based shared task queue is quite high in these tests and shows up vividly in the case of 16 threads. Our implementation can be further improved, but at this point we strive mostly for functional correctness.

Finally, we include performance results for a few simple OpenMP applications. In Fig. 7 we plot timing results for a typical iterative computation of $\pi = 3.14159$, based on the trapezoid rule with 2,000,000 intervals, and using a kernel which spawns a parallel team of 1 to 16 threads. While the scalability is almost ideal, the serial execution is quite slow; the reason is that the Epiphany does not support double-precision numbers natively and the eCORE floating point unit does not implement division. In Fig. 9 we present the performance of a modified version of the Nqueens task benchmark, taken

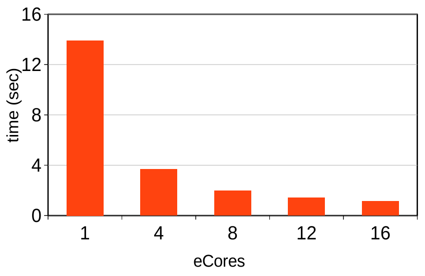

Figure 8: Pi computation

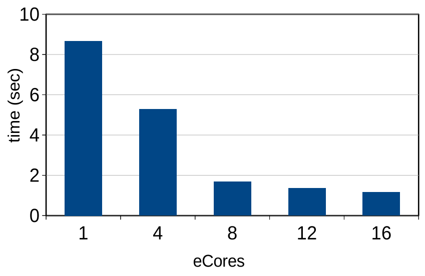

Figure 9: Nqueens(12)

from the Barcelona OpenMP Tasks Suite (BOTS) [10]. This application computes all solutions of the n-queens placement problem on an $n \times n$ chessboard, so that none of the queens threatens any other. Due to the severe memory limitations of the Epiphany, we considered the manual cut-off version of the benchmark, where the nested production of tasks stops at a given depth. The figure shows the timing results for 12 queens, and a cut-off value of 2 (144 tasks produced). As it can be seen, with the addition of eCOREs we obtain an almost linear speedup.

6 Discussion and Current Status

We presented the infrastructure of the OMPi compiler related to the new OpenMP 4.0 device model. We put an emphasis on the efficient handling of data environments both from the compiler and the runtime sides, because the specifications allow considerable freedom in the way data environments can be nested. Our infrastructure is used to support OpenMP within the popular Parallella board, where we treat the Epiphany-16 as an accelerator device, attached to a dual-core ARM host processor, allowing the dynamic creation of parallel teams within the device itself. We are currently concentrated on optimizing our system, and supporting additional OpenMP 4.0 functionality.

From our experimentation with the Parallella board, it became clear that offloaded kernels should not make use of sophisticated OpenMP features, in devices with limited resources. We also observed that a major overhead is the time needed to offload a kernel. If the computation is structured in a way where kernels are repeatedly offloaded, to operate on different data each time, then it is questionable whether the application will experience significant performance gains. For such scenarios, we believe that support of resident kernels may bring considerable improvements. A kernel would be offloaded only once, while at specific points the host would communicate new data (through `target update`s) for the kernel to operate on. Of course, this would also require new synchronization mechanisms between the kernel and the host task, which could be part of future OpenMP extensions.

Acknowledgment.

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

References

[1] Adapteva: Epiphany SDK reference Manual (Sept 2013)

[2] Adapteva: Parallella Reference Manual (Sept 2014)

[3] Agathos, S.N., Papadogiannakis, A., Dimakopoulos, V.V.: Targeting the Parallella. In: Proc. of Euro-Par 2015, to appear. Vienna, Austria (2015)

[4] Appel, A.W.: Modern Compiler Implementation in C. Cambridge University Press, Cambridge (1999)

[5] Bertolli, C., Antao, S.F., Eichenberger, A.E., O'Brien, K., Sura, Z., Jacob, A.C., Chen, T., Sallenave, O.: Coordinating GPU Threads for OpenMP 4.0 in LLVM. In: Proc. of LLVM-HPC '14. pp. 12–21. New Orleans, Louisiana (Nov 2014)

[6] Brown Deer Technology, LLC: COPRTHR API Reference (2014)

[7] Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proc. of 1st EWOMP. pp. 99–105. Lund, Sweden (Sept 1999)

[8] Chunhua, L., Yonghong, Y., de Supinski, B.R., Quinlan, D.J., Chapman, B.M.: Early Experiences with the OpenMP Accelerator Model. In: Proc. of IWOMP 2013. pp. 84–98 (Sept 2013)

[9] Dimakopoulos, V.V., Leontiadis, E., Tzoumas, G.: A portable C compiler for OpenMP V.2.0. In: Proc. of EWOMP 2003. pp. 5–11. Aachen, Germany (Sept 2003)

[10] Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguadé, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In: Proc. of ICPP '09. pp. 124–131. Vienna, Austria (Sept 2009)

[11] GNU: GCC 5 Release Series, https://gcc.gnu.org/gcc-5/changes.html

[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] Khronos OpenCL Working Group: The OpenCL Specification Version: 1.2 (Nov 2012)

[14] Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors, Second Edition: A Hands-on Approach. Morgan Kaufmann, MA 01803, USA (Dec 2012)

[15] Mitra, G., Stotzer, E., Jayaraj, A., Rendell, A.: Implementation and Optimization of the OpenMP Accelerator Model for the TI Keystone II Architecture. In: Proc. of IWOMP 2014, pp. 202–214. Salvador, Brazil (Sept 2014)

[16] Newburn, C., Deodhar, R., Dmitriev, S., Murty, R., Narayanaswamy, R., Wiegert, J., Chinchilla, F., McGuire, R.: Offload Compiler Runtime for the Intel Xeon Phi ™ Coprocessor. In: Proc. of ISC 2013, pp. 239–254. Leipzig, Germany (June 2013)

[17] OpenACC: The OpenACC Application Programming Interface, V.2.0 (June 2013)

[18] OpenMP ARB: OpenMP Application Program Interface V4.0 (July 2013)